Assembly Language

Lecture 4 – Data Transfers, Addressing, and Arithmetic

Ahmed Sallam

Slides based on original lecture slides by Dr. Mahmoud Elgayyar
Outcomes of Lecture 3

• Basic Elements of Assembly Language

• Example: Adding and Subtracting Integers

• Assembling, Linking, and Running Programs

• Defining Data

• Symbolic Constants

• Real-Address Mode Programming
Outline

• Data Transfer Instructions
  ● Operand types
  ● MOV, MOVZX, MOVSX instructions
  ● LAHF, SAHF instructions
  ● XCHG instruction

• Addition and Subtraction
  ● INC and DEC instructions
  ● ADD, SUB instructions
  ● NEG instruction

• Data-Related Operators and Directives

• Indirect Addressing
  ● Arrays and pointers

• JMP and LOOP instructions
Outline

- **Data Transfer Instructions**
  - Operand types
  - MOV, MOVZX, MOVSX instructions
  - LAHF, SAHF instructions
  - XCHG instruction

- **Addition and Subtraction**
  - INC and DEC instructions
  - ADD, SUB instructions
  - NEG instruction

- **Data-Related Operators and Directives**

- **Indirect Addressing**
  - Arrays and pointers

- **JMP and LOOP instructions**
Operand Types

- **Immediate** – a constant integer (8, 16, or 32 bits)
  - value is encoded within the instruction

- **Register** – the name of a register

- **Memory** – reference to a location in memory
  - memory address is encoded within the instruction, or a register holds the address of a memory location

```plaintext
.data
var1 BYTE 10h
;Suppose var1 were located at offset 10400h
mov AL,var1  → A0 00010400
```
## Operand Notation

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

- Move from source to destination
- Syntax:
  \[ \text{MOV destination, source} \]
- Both operands must be the same size
- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment registers moves
- Memory to Memory:
  
  .code
  mov ax,var1
  mov var2,ax
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,[var1] ; AL = 10h
```

**alternate format – Use consistently if you chose to use it**

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

**Use it only when an arithmetic expression is involved**
Mov Errors

.data

    bVal BYTE 100
    bVal2 BYTE ?
    wVal WORD 2
    dVal DWORD 5

.code

    mov al,wVal ; byte <- word
    mov ax,bVal ; word <- byte
    mov eax,bVal ; dword <- byte
    mov ds,45 ; immediate value not permitted
    mov eip,dVal ; invalid destination (eip)
    mov 25,bVal ; invalid destination (25)
    mov bVal2,bVal ; move in mem not permitted
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.

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

The destination must be a register.
• The MOVsx instruction fills the upper half of the destination with a copy of the source operand's sign bit.

\[
\begin{array}{c}
\text{Source} \\
1 0 0 0 1 1 1 1
\end{array}
\begin{array}{c}
\text{Destination} \\
1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1
\end{array}
\]

\begin{verbatim}
mov bl, 10001111b
movsx ax, bl ; sign extension
\end{verbatim}

The destination must be a register.
XCHG Instruction (swap)

- **XCHG exchanges the values of two operands**
- **At least one operand must be a register**
- **No immediate operands are permitted**

```
data
  var1  WORD 1000h
  var2  WORD 2000h
.code
  xchg  ax,bx       ; exchange 16-bit regs
  xchg  ah,al       ; exchange 8-bit regs
  xchg  var1,bx     ; exchange mem, reg
  xchg  eax,ebx     ; exchange 32-bit regs
  xchg  var1,var2   ; error: two memory operands
```
LAHF and SAHF Instructions

- **LAHF: loads status flags into AH**
  - Copies the low byte of the EFLAGS register including Sign, Zero, and Carry flags.
  - 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 into a variable
    ```

- **SAHF: stores AH into status flags**
  - Copies AH into the low byte of the EFLAGS register
  - Retrieve the value of flags stored earlier
    ```
    .code
    mov ah, saveflags ; load save flags into AH
    sahf ; copy into flags register
    ```
So Far...

1. `mov`  move, dest <- source
2. `movzx` move w. zero extend
3. `movsx` move w. sign extend
4. `xchg` swap a register with somewhere else
5. `lahf, sahf` loads/stores flag registers to/from AH

- Don't move memory variables
- Match sizes
Direct-Offset Operands

- A constant offset is added to a data label to produce an effective address (EA)
  - that is de-referenced to get the value inside its memory location

.data

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

.code

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

1. Obtain address specified by label arrayB
2. Add 1 to address (to get second array element)
3. Dereference address to obtain value (20h)
Examples

.data

arrayW    WORD 1000h,2000h,3000h
arrayD    DWORD 1,2,3,4

.code

mov ax,[arrayW+2]       ; AX = 2000h
mov ax,[arrayW+4]       ; AX = 3000h
mov eax,[arrayD+4]      ; EAX = 00000002h
mov ax,[arrayD-2]       ; AX = 3000h
mov eax,[arrayD+20]     ; Possible Seg Fault!

1. There is no "range checking" - the address is calculated and used
2. Size of transfer is based on the destination
Example 2

- **Write a program that adds the following three bytes:**

  ```
  .data
  
  myBytes BYTE 80h, 66h, 0A5h
  ```

- **Solution:**

  ```
  mov al, myBytes
  add al, [myBytes+1]
  add al, [myBytes+2]
  ```
Find the error . . .

- How about the following code. Is anything missing?

```assembly
.data
   myBytes BYTE 80h,66h,0A5h

.code
   movzx ax,myBytes
   mov bl,[myBytes+1]
   add ax,bx
   mov bl,[myBytes+2]
   add ax,bx ; AX = sum
```

What is in bx when we do the add? We loaded bl, what was in bh?
Review Questions

• What are the three basic types of operands?

• (True/False): The destination operand of a MOV instruction cannot be a segment register.

• (True/False): In a MOV instruction, the second operand is known as the destination operand.

• (True/False): The EIP register cannot be the destination operand of a MOV instruction.

• In the operand notation used by Intel, what does reg/mem32 indicate?

• In the operand notation used by Intel, what does imm16 indicate?
Outline

• **Data Transfer Instructions**
  - Operand types
  - MOV, MOVZX, MOVSX instructions
  - LAHF, SAHF instructions
  - XCHG instruction

• **Addition and Subtraction**
  - INC and DEC instructions
  - ADD, SUB instructions
  - NEG instruction

• **Data-Related Operators and Directives**

• **Indirect Addressing**
  - Arrays and pointers

• **JMP and LOOP instructions**
INC and DEC Instructions

• Add 1 or subtract 1 from operand
  - operand may be register or memory

• INC destination
  - Logic: destination ← destination + 1
    (e.g., destination++)

• DEC destination
  - Logic: destination ← destination – 1
    (e.g., destination--)
INC and DEC Examples

.data

    myWord  WORD  1000h
    myDword DWORD 10000000h

.code

    inc myWord       ; 1001h
    dec myWord       ; 1000h
    inc myDword      ; 10000001h

    mov ax,00FFh
    inc ax           ; AX = 0100h
    mov ax,00FFh
    inc al           ; AX = 0000h
ADD and SUB Instructions

• **ADD destination, source**
  - Logic: destination ← destination + source

• **SUB destination, source**
  - Logic: destination ← destination – source

• **Same operand rules as for the MOV instruction**
Examples

.data
    var1    DWORD 10000h
    var2    DWORD 20000h

.code
    mov eax, var1              ; 00010000h
    add eax, var2              ; 00030000h
    add ax, 0FFFFh              ; 0003FFFFh
    add eax, 1                 ; 00040000h
    sub ax, 1                  ; 0004FFFFh

• **Lesson**: You can make yourself really confused and your code becomes garble if you keep using a register for different sized values (ax, al, eax, ah, ax, ...)

• **Pay attention to detail and know exactly what is in every part of a register**
NEG (negate) Instruction

- Reverses the sign of an operand in a register or memory location (2\textsuperscript{nd} complement)

.data
    valB    BYTE  -1
    valW    WORD  +32767

.code
    mov    al,valB       ; AL = -1
    neg    al            ; AL = +1
    neg    valW          ; valW = -32767

- Suppose AX contains \(-32,768\) and we apply NEG to it.

Will the result be valid? Remember, the max positive value is 32767 \((Try\ it!!!)\)
Performing Arithmetic

- HLL compilers translate mathematical expressions into assembly language. You have to do it manually. For example:

\[
Rval = -Xval + (Yval - Zval)
\]

.data
  Rval DWORD ?
  Xval DWORD 26
  Yval DWORD 30
  Zval DWORD 40

.code
  ;first term :-Xval
  mov eax,Xval
  neg eax ; EAX = -26
  ;second term :Yval - Zval
  mov ebx,Yval
  sub ebx,Zval ; EBX = -10
  ;add the terms and store the result
  add eax,ebx
  mov Rval,eax ; -36
The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations based on the contents of the destination operand after the operation.

Essential flags:
- **Zero flag** – set when destination equals zero
- **Sign flag** – set when destination is negative
- **Carry flag** – set when unsigned value is out of range
- **Overflow flag** – set when signed value is out of range.

The **MOV** instruction never affects the flags.
Concept Map

- CPU
  - part of
  - executes

- ALU
  - executes
  - attached to

- status flags
  - affect

- conditional jumps
  - used by
  - provide

- branching logic

arithmetic & bitwise operations
Zero Flag (ZF)

- The Zero flag is set when the result of an operation produces zero in the destination operand.

```
mov cx,1
sub cx,1 ; CX = 0, ZF = 1
mov ax,0FFFFh
add ax,1 ; AX = 0, ZF = 1
add ax,1 ; AX = 1, ZF = 0
```

Remember...

- A flag is **set** when it equals 1
- A flag is **clear** when it equals 0
Sign Flag (SF)

- The Sign flag is set when the destination operand is negative

- The flag is clear when the destination is positive

```
mov cx, 0
sub cx, 1 ; CX = -1, SF = 1
add cx, 2 ; CX =  1, SF = 0
```

- The sign flag is a copy of the destination's highest bit

```
mov al, 0
sub al, 1 ; AL = 11111111b, SF = 1
add al, 2 ; AL = 00000001b, SF = 0
```
Signs on Integers

• All CPU instructions operate exactly the same on signed and unsigned integers

• The CPU cannot distinguish between signed and unsigned integers

• YOU, the programmer, are solely responsible for using the correct data type with each instruction
Overflow and Carry Flags

• How the ADD instruction affects OF and CF:
  - CF = *(carry out)* of the MSB
  - OF = Carry out *XOR* MSB

• How the SUB instruction affects OF and CF:
  - CF = *INVERT* *(carry out)* of the MSB
  - negate the source and add it to the destination
  - OF = Carry out *XOR* MSB

MSB = Most Significant Bit (high-order bit)
XOR = eXclusive-OR operation
NEG = Negate (same as SUB 0,operand)
The Carry flag is set when the result of an operation generates an unsigned value that is out of range.

```assembly
mov al,0FFh
add al,1 ; CF = 1, AL = 00

mov ax,0FFh
add ax,1 ; CF = 0, AX = 0100h
```
Try to go below zero:

```
mov al, 1
sub al, 2 ; CF = 1, AL = FF
```

- The carry out of bit 7 is inverted & placed in CF

INC and DEC instructions doesn’t affect the Carry flag.

Applying the NEG instruction to a nonzero operand always sets the Carry flag. (sub 0, operand)
More Examples

- For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov ax,00FFh

add ax,1 ; AX=0100h, SF=0 ZF=0 CF=0
sub ax,1 ; AX=00FFh, SF=0 ZF=0 CF=0
add al,1 ; AL=00h, SF=0 ZF=1 CF=1

mov bh,6Ch

add bh,95h ; BH=01h, SF=0 ZF=0 CF=1

mov al,2

sub al,3 ; AL=FFh, SF=1 ZF=0 CF=1
```
Overflow Flag (OF)

- The Overflow flag is set when the signed result of an operation is invalid or out of range.

\[
\begin{align*}
\text{mov } al, +127 \\
\text{add } al, 1 & \quad ; \ OF = 1, \ \ AL = -128 \\
\text{mov } al, 7Fh & \quad ; \ OF = 1, \ \ AL = 80h \\
\text{add } al, 1
\end{align*}
\]

- The two examples are identical at the binary level because 

\[
\begin{align*}
7Fh &= +127 \\
80h &= -128
\end{align*}
\]

- To determine the value of the destination operand, it is often easier to calculate in hexadecimal

Lesson: Work in one number system consistently (hex preferably)
Overflow Flag (OF)

- The Overflow flag: The Carry out is exclusive ORed with the high bit of the result.

```assembly
mov al,-128
add al,-2 ; OF = 1, AL = 126
```

![Overflow Flag Diagram]
A Rule of Thumb

- **When adding two integers, remember that the Overflow flag is only set when . . .**
  - Two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

The sign of the result is opposite the sign of the operands

```
What will be the values of the Overflow flag?
mov al,80h ; (-128) + (-110)
add al,92h ; OF = 1, al= 26

mov al,-2
add al,+127 ; OF = 0
```

- Overflow never occurs when the signs of two addition operands are different.

Sub a, b → add a, -b
Warning

- How does the CPU know whether an arithmetic operation is signed or unsigned?
- We can only give what seems a dumb answer: It doesn’t!
- The CPU sets all status flags after an arithmetic operation using a set of Boolean rules,
  - regardless of which flags are relevant.
  - You (the programmer) decide which flags to interpret and which to ignore, based on your knowledge of the type of operation performed.
1. Implement the following expression in assembly language: \( AX = (-val2 + BX) - val4. \)

2. (Yes/No): Is it possible to set the Overflow flag if you add a positive integer to a negative integer?

3. (Yes/No): Is it possible for the NEG instruction to set the Overflow flag?

4. (Yes/No): Is it possible for both the Sign and Zero flags to be set at the same time?

5. Write a sequence of two instructions that set both the Carry and Overflow flags at the same time.

6. Write a sequence of instructions showing how the Zero flag could be used to indicate unsigned overflow (carry flag) after executing the INC instruction.
Outline

• **Data Transfer Instructions**
  - Operand types
  - MOV, MOVZX, MOVSX instructions
  - LAHF, SAHF instructions
  - XCHG instruction

• **Addition and Subtraction**
  - INC and DEC instructions
  - ADD, SUB instructions
  - NEG instruction

• **Data-Related Operators and Directives**

• **Indirect Addressing**
  - Arrays and pointers

• **JMP and LOOP instructions**
OFFSET Operator

- OFFSET returns the distance in bytes of a label from the beginning of its enclosing segment

  - Protected mode: 32 bits
  - Real mode: 16 bits

The Protected-Mode programs we write use only a single data segment due to the flat memory model.
OFFSET Examples

• Assume that \textit{bVal} were located at offset 00404000h:

\begin{verbatim}
.data
  bVal  BYTE  ?
  wVal  WORD  ?
  dVal  DWORD  ?
  dVal2 DWORD  ?

.code
  mov esi,OFFSET bVal  ; ESI = 00404000
  mov esi,OFFSET wVal  ; ESI = 00404001
  mov esi,OFFSET dVal  ; ESI = 00404003
  mov esi,OFFSET dVal2 ; ESI = 00404007
\end{verbatim}
// C++ version:
char array[1000];
char * p = array;

; Assembly language:
.data
array BYTE 1000 DUP(?)
.code
mov esi,OFFSET array
The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary:

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

- **Overrides the default type of a label (variable).**
- **Provides the flexibility to access part of a variable**
- **Requires a prefixed size specifier**

```assembly
.data
    myDouble DWORD 12345678h
.code
    mov ax,myDouble            ;error! word<-dword
    mov ax,WORD PTR myDouble   ;loads 5678h
    mov WORD PTR myDouble,4321h ;saves 4321h
```
Little Endian Order (again)

- Little endian order refers to the way Intel stores integers in memory
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the DWORD 12345678h would be stored as:

<table>
<thead>
<tr>
<th>byte</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>0000</td>
</tr>
<tr>
<td>56</td>
<td>0001</td>
</tr>
<tr>
<td>34</td>
<td>0002</td>
</tr>
<tr>
<td>12</td>
<td>0003</td>
</tr>
</tbody>
</table>

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.
.data
myDouble DWORD 12345678h

<table>
<thead>
<tr>
<th>doubleword</th>
<th>word</th>
<th>byte</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>5678</td>
<td>78</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>1234</td>
<td>34</td>
<td>12</td>
<td>0002</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0003</td>
</tr>
</tbody>
</table>

mov al, BYTE PTR myDouble ; AL = 78h
mov al, BYTE PTR [myDouble+1] ; AL = 56h
mov al, BYTE PTR [myDouble+2] ; AL = 34h
mov ax, WORD PTR myDouble ; AX = 5678h
mov ax, WORD PTR [myDouble+2] ; AX = 1234h
Joining Words

- **PTR can also be used to combine elements of a smaller data type and move them into a larger operand**

- **The CPU will automatically reverse the bytes**

```assembly
.data
    myBytes BYTE 12h,34h,56h,78h

.code
    mov ax,WORD PTR [myBytes] ; AX = 3412h
    mov ax,WORD PTR [myBytes+2] ; AX = 7856h
    mov eax,DWORD PTR myBytes ; EAX = 78563412h
```
.data
    varB BYTE 65h,31h,02h,05h
    varW WORD 6543h,1202h
    varD DWORD 12345678h

.code
    mov ax,WORD PTR [varB+2] ; ax=0502h
    mov bl,BYTE PTR varD     ; bl=78h
    mov bl,BYTE PTR [varW+2] ; bl=02h
    mov ax,WORD PTR [varD+2] ; ax=1234h
    mov eax,DWORD PTR varW   ; eax=12026543h
The TYPE operator returns the size in bytes of a single element of a data declaration.

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

.code
    mov eax,TYPE var1   ; 1
    mov eax,TYPE var2   ; 2
    mov eax,TYPE var3   ; 4
    mov eax,TYPE var4   ; 8
```
The `LENGTHOF` operator counts the number of elements in a single data declaration.

```assembly
.data
    byte1     BYTE 10,20,30 ; 3
    array1    WORD 30 DUP(?),0,0 ; 32
    array2    WORD 5 DUP(3 DUP(?)) ; 15
    array3    DWORD 1,2,3,4 ; 4
    digitStr  BYTE "12345678",0 ; 9

.code
    mov ecx,LENGTHOF array1 ; 32
```
**SIZEOF Operator**

- The `SIZEOF` operator is equivalent to multiplying:

  \[ \text{SIZEOF} = \text{LENGTHOF} \times \text{TYPE} \]

```
.data

byte1 BYTE 10,20,30 ; 3
array1 WORD 30 DUP(?),0,0 ; 64
array2 WORD 5 DUP(3 DUP(?)) ; 30
array3 DWORD 1,2,3,4 ; 16
digitStr BYTE "12345678",0 ; 9

.code

mov ecx,SIZEOF array1 ; 64
```
Summary

- **OFFSET**: Distance from beginning of data segment (i.e., a partial address)
- **PTR**: Changes the size of a value (i.e., a cast)
- **TYPE**: Size in bytes of a value
- **LENGTHOF**: Number of data elements
- **SIZEOF**: TYPE * LENGTHOF (i.e., total bytes used)
Spanning Multiple Lines

- A data declaration spans multiple lines if each line (except the last) ends with a comma

- The LENGTHOF and SIZEOF operators include all lines belonging to the declaration

```assembly
.data
array    WORD  10,20,
         30,40,
         50,60

.code
  mov eax,LENGTHOF array    ; eax=6
  mov ebx,SIZEOF array      ; eax=12
```
Contrast: Anonymous Data

- In the following example, array identifies only the first WORD declaration, with 2 values, even though the name can be used to access all 6 words.

- SIZEOF/LENGTHOF are assembly directives, NOT runtime instructions.

```assembly
.data
array   WORD 10,20    ; array ends here
         WORD 30,40    ; anonymous data, array+4
         WORD 50,60    ; array+8

.code
     mov eax,LENGTHOF array    ; 2
     mov ebx,SIZEOF array      ; 4
```
LABEL Directive

- Assigns an alternate label name and type to an existing storage location
- LABEL does not allocate any storage of its own
- Avoids the need for the PTR operator

```
.data
dwList   LABEL DWORD
wordList LABEL WORD
intList  BYTE 00h,10h,00h,20h

.code
mov eax,dwList    ; 20001000h
mov cx,wordList    ; 1000h
mov dl,intList     ; 00h
```

dwList, wordList, intList are the same offset (address)
Review Questions

1. (True/False): The OFFSET operator always returns a 16-bit value.
2. (True/False): The PTR operator returns the 32-bit address of a variable.
3. (True/False): The TYPE operator returns a value of 4 for doubleword operands.
4. (True/False): The LENGTHOF operator returns the number of bytes in an operand.
5. (True/False): The SIZEOF operator returns the number of bytes in an operand.
Outline

• *Data Transfer Instructions*
  - Operand types
  - MOV, MOVZX, MOVSX instructions
  - LAHF, SAHF instructions
  - XCHG instruction

• *Addition and Subtraction*
  - INC and DEC instructions
  - ADD, SUB instructions
  - NEG instruction

• *Data-Related Operators and Directives*

• *Indirect Addressing*
  - Arrays and pointers

• *JMP and LOOP instructions*
Indirect Operands

- An indirect operand holds the address of a variable, usually an array or string

- It can be de-referenced (just like a pointer) using [ and ]

- Works with OFFSET to produce the address to de-reference

```assembly
.data
  val1 BYTE 10h,20h,30h
.code
  mov esi,OFFSET val1 ; esi stores address of val1
  mov al, [esi] ; dereference ESI (AL = 10h)

  inc esi
  mov al, [esi] ; AL = 20h

  inc esi
  mov al, [esi] ; AL = 30h
```

NOTE: We tend to use esi and edi to store addresses
Using PTR

- Use PTR to clarify the size attribute of a memory operand

- When we have an address (offset) we don't know the size of the values at that offset and must specify them explicitly

```
.data
    myCount WORD 0

.code
    mov esi,OFFSET myCount

    inc [esi] ; error: operand must have size
    inc WORD PTR [esi] ; ok

    add [esi],20 ; error:...
    add WORD PTR [esi],20 ; ok
```
Array Sum Example

- Indirect operands are ideal for traversing an array

- Note: the register in brackets must be incremented by a value that matches the array TYPE (i.e., 2 for WORD, 4 for DWORD, 8 for QWORD)

.data

    arrayW WORD 1000h,2000h,3000h

.code

    mov esi,OFFSET arrayW
    mov ax,[esi]
    add esi,2
    ;or   add esi,TYPE arrayW   ; good clarity
    add ax,[esi]
    add esi,2
    add ax,[esi]               ; AX = sum of the array
Indexed Operands

- An indexed operand adds an address and a register to generate an effective address.

- There are two notational forms:
  - `[label + reg]`
  - `label[reg]`

```
.data
arrayW WORD 1000h,2000h,3000h

.code
mov esi,0
mov ax,[arrayW + esi] ; AX = 1000h
mov ax,arrayW[esi] ; alternate format
add esi,TYPE arrayW
add ax,[arrayW + esi]
```
Index Scaling

- **You can scale an indirect or indexed operand to the offset of an array element by multiplying the index by the array's TYPE:**

```
data
    arrayB BYTE 0,1,2,3,4,5
    arrayW WORD 0,1,2,3,4,5
    arrayD DWORD 0,1,2,3,4,5

.code
    mov esi,4
    mov al,arrayB[esi*TYPE arrayB] ; 04
    mov bx,arrayW[esi*TYPE arrayW] ; 0004
    mov edx,arrayD[esi*TYPE arrayD] ; 00000004
```
Pointers

• **Offsets are of size DWORD**

• **A variable of size DWORD can hold an offset**

• **i.e., you can declare a pointer variable that contains the offset of another variable.**

```
.data
  arrayW WORD 1000h,2000h,3000h
  ptrW DWORD arrayW  ; ptrW = offset of arrayW
  ; Alternative – same as above
  ; ptrW DWORD OFFSET arrayW

.code
  mov esi,ptrW

  mov ax,[esi]  ; AX = 1000h
```
1. (True/False): Any 32-bit general-purpose register can be used as an indirect operand.

2. (True/False): The BX register is usually reserved for addressing the stack.

3. (True/False): The following instruction is invalid: inc [esi]

4. (True/False): The following is an indexed operand: array[esi]
Outline

• **Data Transfer Instructions**
  - Operand types
  - MOV, MOVZX, MOVSX instructions
  - LAHF, SAHF instructions
  - XCHG instruction

• **Addition and Subtraction**
  - INC and DEC instructions
  - ADD, SUB instructions
  - NEG instruction

• **Data-Related Operators and Directives**

• **Indirect Addressing**
  - Arrays and pointers

• **JMP and LOOP instructions**
JMP Instruction

- Jumps are the basis of most control flow
- HLL compilers turn loops, if statements, switches etc. into some kind of jump
- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: JMP target
- Logic: EIP ← target

Example

top:
  .
  .
jmp top

A jump outside the current procedure must be to a special type of label called a global label (which we will examine when we examine procedures)
The LOOP instruction creates a **counted loop** using ECX

**Syntax:** LOOP target

**target should precede the instruction**
- ECX must contain the iteration count

**Logic:**
- ECX ← ECX – 1
- if ECX ≠ 0, jump back to target, else go to next instruction

```assembly
mov ax, 0
mov ecx, 5
L1:
    add ax, cx
    loop L1
```

This loop calculates the sum: $5 + 4 + 3 + 2 + 1$
Examples

```
mov ax, 6
mov ecx, 4 ; Loop 4 times
L1:
  inc ax ; Each iteration ax++ (7, 8, 9, 10)
loop L1
```

```
mov ecx, 0 ; ecx starts at 0! (an error)
X2:
  inc ax ; ax++ until ecx holds 0
loop X2 ; ecx− (-1, -2, -3, . . . )

; ax = 4294967296 when you exit the loop
```
Nested Loops

- **If you need to code a loop within a loop, you must save the outer loop counter's ECX value**

- **In this example, the outer loop executes 100 times, and the inner loop 20 times**

```assembly
.data
    count DWORD ?
.code
    mov ecx,100 ; set outer loop count
L1:
    mov count,ecx ; save outer loop count
    mov ecx,20 ; set inner loop count
L2:
.loop L2 ; repeat the inner loop
    mov ecx,count ; restore outer loop count
    loop L1 ; repeat the outer loop
```
Summing an Array

.data

intarray WORD 100h,200h,300h,400h

.code

mov edi,OFFSET intarray ; address of intarray
mov ecx,LENGTHOF intarray ; loop counter
mov ax,0 ; zero the accumulator

L1:

add ax,[edi] ; add an integer
add edi,TYPE intarray ; point to next integer

loop L1 ; repeat until ECX = 0
Copying a String

.data
    source BYTE "This is the source string",0
    target BYTE SIZEOF source DUP(0)

.code
    mov esi,0 ; index register
    mov ecx,SIZEOF source ; loop counter
L1:
    mov al,source[esi] ; get char from source
    mov target[esi],al ; store it in the target
    inc esi ; move to next character
    loop L1 ; repeat for entire string
Review Questions

1. (True/False): A JMP instruction can only jump to a label inside the current procedure.

2. (True/False): The LOOP instruction first checks to see whether ECX is not equal to zero; then LOOP decrements ECX and jumps to the destination label.

3. (Challenge): What will be the final value of EAX in this example?
   
   ```
   mov eax,0
   mov ecx,10 ; outer loop counter
   L1:
   mov eax,3
   mov ecx,5 ; inner loop counter
   L2:
   add eax,5
   loop L2 ; repeat inner loop
   loop L1 ; repeat outer loop
   ```

4. Revise the code from the preceding question so the outer loop counter is not erased when the inner loop starts.
Summary

• **Data Transfer**
  - MOV – data transfer from source to destination
  - MOVSX, MOVZ, XCHG

• **Operand types**
  - direct, direct-offset, indirect, indexed

• **Arithmetic**
  - INC, DEC, ADD, SUB, NEG
  - Sign, Carry, Zero, Overflow flags

• **Operators**
  - OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, TYPEDEF

• **JMP and LOOP – branching instructions**