

Basic Instructions Addressing Modes

COE 205

Computer Organization and Assembly Language

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Presentation Outline

- ❖ **Operand Types**
- ❖ Data Transfer Instructions
- ❖ Addition and Subtraction
- ❖ Addressing Modes
- ❖ Jump and Loop Instructions
- ❖ Copying a String
- ❖ Summing an Array of Integers

Three Basic Types of Operands

❖ Immediate

- ❖ Constant integer (8, 16, or 32 bits)
- ❖ Constant value is stored within the instruction

❖ Register

- ❖ Name of a register is specified
- ❖ Register number is encoded within the instruction

❖ Memory

- ❖ Reference to a location in memory
- ❖ Memory address is encoded within the instruction, or
- ❖ Register holds the address of a memory location

Instruction Operand Notation

Operand	Description
<i>r8</i>	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
<i>r16</i>	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
<i>r32</i>	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
<i>reg</i>	Any general-purpose register
<i>sreg</i>	16-bit segment register: CS, DS, SS, ES, FS, GS
<i>imm</i>	8-, 16-, or 32-bit immediate value
<i>imm8</i>	8-bit immediate byte value
<i>imm16</i>	16-bit immediate word value
<i>imm32</i>	32-bit immediate doubleword value
<i>r/m8</i>	8-bit operand which can be an 8-bit general-purpose register or memory byte
<i>r/m16</i>	16-bit operand which can be a 16-bit general-purpose register or memory word
<i>r/m32</i>	32-bit operand which can be a 32-bit general register or memory doubleword
<i>mem</i>	8-, 16-, or 32-bit memory operand

Next ...

- ❖ Operand Types
- ❖ **Data Transfer Instructions**
- ❖ Addition and Subtraction
- ❖ Addressing Modes
- ❖ Jump and Loop Instructions
- ❖ Copying a String
- ❖ Summing an Array of Integers

MOV Instruction

- ❖ Move source operand to destination
`mov destination, source`
- ❖ Source and destination operands can vary
 - `mov reg, reg`
 - `mov mem, reg`
 - `mov reg, mem`
 - `mov mem, imm`
 - `mov reg, imm`
 - `mov r/m16, sreg`
 - `mov sreg, r/m16`
- ❖ Programs running in protected mode should not modify the segment registers

Rules

- Both operands must be of same size
- No memory to memory moves
- Destination cannot be CS, EIP, or IP
- No immediate to segment moves

MOV Examples

```
.DATA
count BYTE 100
bVal  BYTE 20
wVal  WORD 2
dVal  DWORD 5

.CODE

mov bl, count ; bl = count = 100
mov ax, wVal  ; ax = wVal = 2
mov count,al  ; count = al = 2
mov eax, dval ; eax = dval = 5

; Assembler will not accept the following moves - why?
mov ds, 45 ; immediate move to DS not permitted
mov esi, wVal ; size mismatch
mov eip, dVal ; EIP cannot be the destination
mov 25, bVal ; immediate value cannot be destination
mov bVal, count ; memory-to-memory move not permitted
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

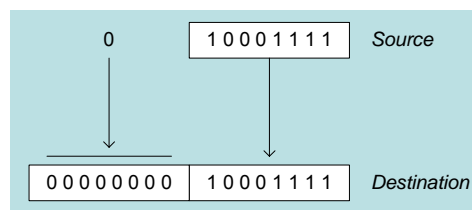
© Muhamed Mudawar – slide 7

Zero Extension

❖ MOVZX Instruction

- ❖ Fills (extends) the upper part of the destination with zeros
- ❖ Used to copy a small source into a larger destination
- ❖ Destination must be a register

```
movzx r32, r/m8
movzx r32, r/m16
movzx r16, r/m8
```



```
mov  bl, 8Fh
movzx ax, bl
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 8

Sign Extension

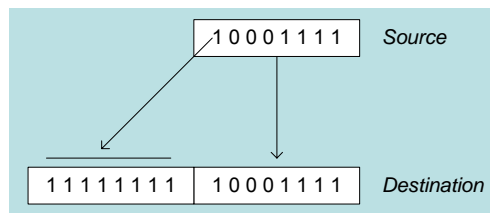
❖ MOVSX Instruction

- ❖ Fills (extends) the upper part of the destination register with a copy of the source operand's sign bit
- ❖ Used to copy a small source into a larger destination

```
movsx r32, r/m8
```

```
movsx r32, r/m16
```

```
movsx r16, r/m8
```



```
mov  bl, 8Fh  
movsx ax, bl
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 9

XCHG Instruction

❖ XCHG exchanges the values of two operands

```
xchg reg, reg
```

```
xchg reg, mem
```

```
xchg mem, reg
```

Rules

- Operands must be of the same size
- At least one operand must be a register
- No immediate operands are permitted

```
.DATA  
var1 DWORD 10000000h  
var2 DWORD 20000000h  
.CODE  
xchg ah, al    ; exchange 8-bit regs  
xchg ax, bx    ; exchange 16-bit regs  
xchg eax, ebx  ; exchange 32-bit regs  
xchg var1, ebx ; exchange mem, reg  
xchg var1, var2 ; error: two memory operands
```

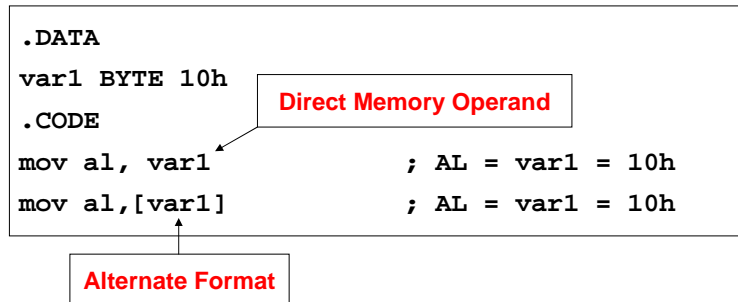
Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 10

Direct Memory Operands

- ❖ Variable names are references to locations in memory
- ❖ Direct Memory Operand:
 - Named reference to a memory location
- ❖ Assembler computes address (offset) of named variable



Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 11

Direct-Offset Operands

- ❖ Direct-Offset Operand: Constant offset is added to a named memory location to produce an **effective address**
 - ❖ Assembler computes the **effective address**
- ❖ Lets you access memory locations that have **no name**

```
.DATA
arrayB BYTE 10h,20h,30h,40h
.CODE
mov al, arrayB+1      ; AL = 20h
mov al,[arrayB+1]    ; alternative notation
mov al, arrayB[1]     ; yet another notation
```

Q: Why doesn't `arrayB+1` produce 11h?

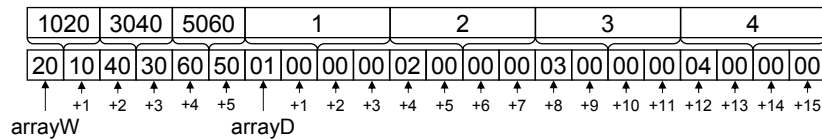
Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 12

Direct-Offset Operands - Examples

```
.DATA
arrayW WORD 1020h, 3040h, 5060h
arrayD DWORD 1, 2, 3, 4
.CODE
mov ax, arrayW+2 ; AX = 3040h
mov ax, arrayW[4] ; AX = 5060h
mov eax,[arrayD+4] ; EAX = 0000002h
mov eax,[arrayD-3] ; EAX = 01506030h
mov ax, [arrayW+9] ; AX = 0200h
mov ax, [arrayD+3] ; Error: Operands are not same size
mov ax, [arrayW-2] ; AX = ? Out-of-range address
mov eax,[arrayD+16] ; EAX = ? MASM does not detect error
```



Basic Instructions & Addressing Modes

COE 205 - KFUPM

© Muhamed Mudawar - slide 13

Your Turn ...

Given the following definition of arrayD

```
.DATA
arrayD DWORD 1,2,3
```

Rearrange the three values in the array as: 3, 1, 2

Solution:

```
; Copy first array value into EAX
mov eax, arrayD ; EAX = 1
; Exchange EAX with second array element
xchg eax, arrayD[4] ; EAX = 2, arrayD = 1,1,3
; Exchange EAX with third array element
xchg eax, arrayD[8] ; EAX = 3, arrayD = 1,1,2
; Copy value in EAX to first array element
mov arrayD, eax ; arrayD = 3,1,2
```

Basic Instructions & Addressing Modes

COE 205 - KFUPM

© Muhamed Mudawar - slide 14

Next ...

- ❖ Operand Types
- ❖ Data Transfer Instructions
- ❖ **Addition and Subtraction**
- ❖ Addressing Modes
- ❖ Jump and Loop Instructions
- ❖ Copying a String
- ❖ Summing an Array of Integers

ADD and SUB Instructions

- ❖ ADD *destination, source*
 $destination = destination + source$
- ❖ SUB *destination, source*
 $destination = destination - source$
- ❖ Destination can be a **register** or a **memory** location
- ❖ Source can be a **register**, **memory** location, or a **constant**
- ❖ Destination and source must be of the **same size**
- ❖ Memory-to-memory arithmetic is not allowed

Evaluate this . . .

Write a program that adds the following three words:

```
.DATA
array WORD 890Fh,1276h,0AF5Bh
```

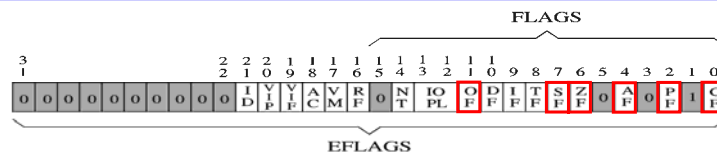
Solution: Accumulate the sum in the AX register

```
mov ax, array
add ax,[array+2]
add ax,[array+4] ; what if sum cannot fit in AX?
```

Solution 2: Accumulate the sum in the EAX register

```
movzx eax, array ; error to say: mov eax,array
movzx ebx, array[2] ; use movsx for signed integers
add eax, ebx ; error to say: add eax,array[2]
movzx ebx, array[4]
add eax, ebx
```

Flags Affected



ADD and SUB affect all the six status flags:

1. Carry Flag: Set when **unsigned** arithmetic result is out of range
2. Overflow Flag: Set when **signed** arithmetic result is out of range
3. Sign Flag: Copy of **sign bit**, set when result is **negative**
4. Zero Flag: Set when result is **zero**
5. Auxiliary Carry Flag: Set when there is a **carry from bit 3 to bit 4**
6. Parity Flag: Set when parity in least-significant byte is **even**

More on Carry and Overflow

❖ Addition: $A + B$

- ❖ The Carry flag is the carry out of the most significant bit
- ❖ 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
 - Overflow cannot occur when adding operands of opposite signs

❖ Subtraction: $A - B$

- ❖ For Subtraction, the carry flag becomes the **borrow flag**
- ❖ Carry flag is set when A has a smaller unsigned value than B
- ❖ The Overflow flag is only set when . . .
 - A and B have different signs and sign of result \neq sign of A
 - Overflow cannot occur when subtracting operands of the same sign

Hardware Viewpoint

❖ CPU cannot distinguish signed from unsigned integers

- ❖ YOU, the programmer, give a meaning to binary numbers

❖ How the ADD instruction modifies OF and CF:

- ❖ $CF = (\text{carry out of the MSB})$ ← **MSB = Most Significant Bit**

- ❖ $OF = (\text{carry out of the MSB}) \text{ XOR } (\text{carry into the MSB})$

❖ Hardware does SUB by ...

↑ **XOR = eXclusive-OR operation**

- ❖ ADDing destination to the 2's complement of the source operand

❖ How the SUB instruction modifies OF and CF:

- ❖ Negate (2's complement) the source and ADD it to destination

- ❖ $OF = (\text{carry out of the MSB}) \text{ XOR } (\text{carry into the MSB})$

- ❖ $CF = \text{INVERT} (\text{carry out of the MSB})$

ADD and SUB Examples

For each of the following marked entries, show the values of the destination operand and the six status flags:

```

mov al,0FFh ; AL=-1
add al,1 ; AL=00h CF=1 OF=0 SF=0 ZF=1 AF=1 PF=1
sub al,1 ; AL=FFh CF=1 OF=0 SF=1 ZF=0 AF=1 PF=1
mov al,+127 ; AL=7Fh
add al,1 ; AL=-128 CF=0 OF=1 SF=1 ZF=0 AF=1 PF=0
mov al,26h
sub al,95h ; AL=91h CF=1 OF=1 SF=1 ZF=0, AF=0 PF=0
    
```

1	0	0	1	0	0	0	1	0	1	1	0	1	1	0			
	0	0	1	0	0	1	1	0	+	0	0	1	0	0	1	1	0
-	1	0	0	1	0	1	0	1		0	1	1	0	1	0	1	1
	1	0	0	1	0	0	0	1		1	0	0	1	0	0	0	1

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 21

INC, DEC, and NEG Instructions

- ❖ **INC destination**
 - ❖ $destination = destination + 1$
 - ❖ More compact (uses less space) than: **ADD destination, 1**
- ❖ **DEC destination**
 - ❖ $destination = destination - 1$
 - ❖ More compact (uses less space) than: **SUB destination, 1**
- ❖ **NEG destination**
 - ❖ $destination = 2's\ complement\ of\ destination$
- ❖ Destination can be 8-, 16-, or 32-bit operand
 - ❖ In memory or a register
 - ❖ NO immediate operand

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 22

Affected Flags

- ❖ INC and DEC affect five status flags
 - ✧ Overflow, Sign, Zero, Auxiliary Carry, and Parity
 - ✧ Carry flag is NOT modified
- ❖ NEG affects all the six status flags
 - ✧ Any nonzero operand causes the carry flag to be set

```
.DATA
  B SBYTE -1      ; 0FFh
  C SBYTE 127    ; 7Fh
.CODE
  inc B ; B=0      OF=0 SF=0 ZF=1 AF=1 PF=1
  dec B ; B=-1     OF=0 SF=1 ZF=0 AF=1 PF=1
  inc C ; C=-128=80h OF=1 SF=1 ZF=0 AF=1 PF=0
  neg C ; W=-128  CF=1 OF=1 SF=1 ZF=0 AF=0 PF=0
```

ADC and SBB Instruction

- ❖ ADC Instruction: Addition with Carry
 - ADC destination, source**
 - $destination = destination + source + CF$
- ❖ SBB Instruction: Subtract with Borrow
 - SBB destination, source**
 - $destination = destination - source - CF$
- ❖ Destination can be a **register** or a **memory** location
- ❖ Source can be a **register**, **memory** location, or a **constant**
- ❖ Destination and source must be of the **same size**
- ❖ Memory-to-memory arithmetic is not allowed

Extended Arithmetic

- ❖ ADC and SBB are useful for extended arithmetic
- ❖ Example: 64-bit addition
 - ◇ Assume first 64-bit integer operand is stored in EBX:EAX
 - ◇ Second 64-bit integer operand is stored in EDX:ECX

- ❖ **Solution:**

```
add eax, ecx ;add lower 32 bits
```

```
adc ebx, edx ;add upper 32 bits + carry
```

64-bit result is in EBX:EAX

- ❖ STC and CLC Instructions
 - ◇ Used to Set and Clear the Carry Flag

Next ...

- ❖ Operand Types
- ❖ Data Transfer Instructions
- ❖ Addition and Subtraction
- ❖ **Addressing Modes**
- ❖ Jump and Loop Instructions
- ❖ Copying a String
- ❖ Summing an Array of Integers

Addressing Modes

- ❖ Two Basic Questions
 - ◇ Where are the operands?
 - ◇ How memory addresses are computed?
- ❖ Intel IA-32 supports 3 fundamental addressing modes
 - ◇ **Register** addressing: operand is in a register
 - ◇ **Immediate** addressing: operand is stored in the instruction itself
 - ◇ **Memory** addressing: operand is in memory
- ❖ Memory Addressing
 - ◇ Variety of addressing modes
 - ◇ Direct and indirect addressing
 - ◇ Support high-level language constructs and data structures

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 27

Register and Immediate Addressing

- ❖ Register Addressing
 - ◇ Most efficient way of specifying an operand: no memory access
 - ◇ Shorter Instructions: fewer bits are needed to specify register
 - ◇ Compilers use registers to optimize code
- ❖ Immediate Addressing
 - ◇ Used to specify a constant
 - ◇ Immediate constant is part of the instruction
 - ◇ Efficient: no separate operand fetch is needed

❖ Examples

```
mov eax, ebx    ; register-to-register move
add eax, 5      ; 5 is an immediate constant
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 28

Direct Memory Addressing

- ❖ Used to address simple variables in memory
 - ❖ Variables are defined in the data section of the program
 - ❖ We use the variable name (label) to address memory directly
 - ❖ Assembler computes the offset of a variable
 - ❖ The variable offset is specified directly as part of the instruction

❖ Example

```
.data
var1  DWORD  100
var2  DWORD  200
sum   DWORD  ?
.code
mov  eax, var1
add  eax, var2
mov  sum, eax
```

var1, var2, and sum are direct memory operands

Register Indirect Addressing

- ❖ Problem with Direct Memory Addressing
 - ❖ Causes problems in addressing arrays and data structures
 - Does not facilitate using a loop to traverse an array
 - ❖ Indirect memory addressing solves this problem
- ❖ Register Indirect Addressing
 - ❖ The memory address is stored in a register
 - ❖ Brackets [] used to surround the register holding the address
 - ❖ For 32-bit addressing, any 32-bit register can be used

❖ Example

```
mov  ebx, OFFSET array ; ebx contains the address
mov  eax, [ebx]         ; [ebx] used to access memory
```

EBX contains the **address** of the operand, not the operand itself

Array Sum Example

- ❖ Indirect addressing is ideal for traversing an array

```
.data
    array DWORD 10000h,20000h,30000h
.code
    mov esi, OFFSET array    ; esi = array address
    mov eax,[esi]            ; eax = [array] = 10000h
    add esi,4                ; why 4?
    add eax,[esi]            ; eax = eax + [array+4]
    add esi,4                ; why 4?
    add eax,[esi]            ; eax = eax + [array+8]
```

- ❖ Note that ESI register is used as a **pointer** to array
 - ❖ ESI must be incremented by 4 to access the next array element
 - Because each array element is 4 bytes (DWORD) in memory

Ambiguous Indirect Operands

- ❖ Consider the following instructions:

```
mov [EBX], 100
add [ESI], 20
inc [EDI]
```

- ❖ Where EBX, ESI, and EDI contain memory addresses
- ❖ The size of the memory operand is not clear to the assembler
 - EBX, ESI, and EDI can be pointers to BYTE, WORD, or DWORD

- ❖ **Solution:** use **PTR** operator to clarify the operand size

```
mov BYTE PTR [EBX], 100 ; BYTE operand in memory
add WORD PTR [ESI], 20  ; WORD operand in memory
inc DWORD PTR [EDI]    ; DWORD operand in memory
```


Indexed Addressing

- ❖ Combines a variable's name with an index register
 - ❖ Assembler converts variable's name into a **constant offset**
 - ❖ Constant offset is added to register to form an **effective address**
- ❖ Syntax: $[name + index]$ or $name [index]$

```
.data
    array DWORD 10000h,20000h,30000h
.code
    mov esi, 0           ; esi = array index
    mov eax,array[esi]   ; eax = array[0] = 10000h
    add esi,4
    add eax,array[esi]   ; eax = eax + array[4]
    add esi,4
    add eax,[array+esi]  ; eax = eax + array[8]
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 33

Index Scaling

- ❖ Useful to index array elements of size 2, 4, and 8 bytes
 - ❖ Syntax: $[name + index * scale]$ or $name [index * scale]$
- ❖ Effective address is computed as follows:
 - ❖ Name's offset + Index register * Scale factor

```
.DATA
    arrayB BYTE 10h,20h,30h,40h
    arrayW WORD 100h,200h,300h,400h
    arrayD DWORD 10000h,20000h,30000h,40000h
.CODE
    mov esi, 2
    mov al, arrayB[esi]   ; AL = 30h
    mov ax, arrayW[esi*2] ; AX = 300h
    mov eax, arrayD[esi*4] ; EAX = 30000h
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 34

Based Addressing

- ❖ Syntax: $[Base + Offset]$
 - ◇ Effective Address = Base register + Constant Offset
- ❖ Useful to access fields of a structure or an object
 - ◇ Base Register → points to the base address of the structure
 - ◇ Constant Offset → relative offset within the structure

```
.DATA
mystruct WORD 12
          DWORD 1985
          BYTE 'M'

.CODE
mov ebx, OFFSET mystruct
mov eax, [ebx+2]           ; EAX = 1985
mov al, [ebx+6]           ; AL = 'M'
```

mystruct is a structure consisting of 3 fields: a word, a double word, and a byte

Based-Indexed Addressing

- ❖ Syntax: $[Base + (Index * Scale) + Offset]$
 - ◇ Scale factor is optional and can be 1, 2, 4, or 8
- ❖ Useful in accessing two-dimensional arrays
 - ◇ Offset: array address => we can refer to the array by name
 - ◇ Base register: holds row address => relative to start of array
 - ◇ Index register: selects an element of the row => column index
 - ◇ Scaling factor: when array element size is 2, 4, or 8 bytes
- ❖ Useful in accessing arrays of structures (or objects)
 - ◇ Base register: holds the address of the array
 - ◇ Index register: holds the element address relative to the base
 - ◇ Offset: represents the offset of a field within a structure

Based-Indexed Examples

```
.data
matrix  DWORD  0, 1, 2, 3, 4 ; 4 rows, 5 cols
        DWORD 10,11,12,13,14
        DWORD 20,21,22,23,24
        DWORD 30,31,32,33,34

ROWSIZE EQU  SIZEOF matrix ; 20 bytes per row

.code
mov ebx, 2*ROWSIZE ; row index = 2
mov esi, 3         ; col index = 3
mov eax, matrix[ebx+esi*4] ; EAX = matrix[2][3]

mov ebx, 3*ROWSIZE ; row index = 3
mov esi, 1         ; col index = 1
mov eax, matrix[ebx+esi*4] ; EAX = matrix[3][1]
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 37

LEA Instruction

❖ LEA = Load Effective Address

LEA r32, mem (Flat-Memory)

LEA r16, mem (Real-Address Mode)

- ❖ Calculate and load the effective address of a memory operand
- ❖ Flat memory uses 32-bit effective addresses
- ❖ Real-address mode uses 16-bit effective addresses

❖ LEA is similar to MOV ... OFFSET, except that:

- ❖ OFFSET **operator** is executed by the **assembler**
 - Used with named variables: address is known to the assembler
- ❖ LEA **instruction** computes effective address **at runtime**
 - Used with indirect operands: effective address is known at runtime

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 38

LEA Examples

```

.data
    array WORD 1000 DUP(?)

.code
    lea eax, array           ; Equivalent to . . .
                           ; mov eax, OFFSET array

    lea eax, array[esi]     ; mov eax, esi
                           ; add eax, OFFSET array

    lea eax, array[esi*2]   ; mov eax, esi
                           ; add eax, eax
                           ; add eax, OFFSET array

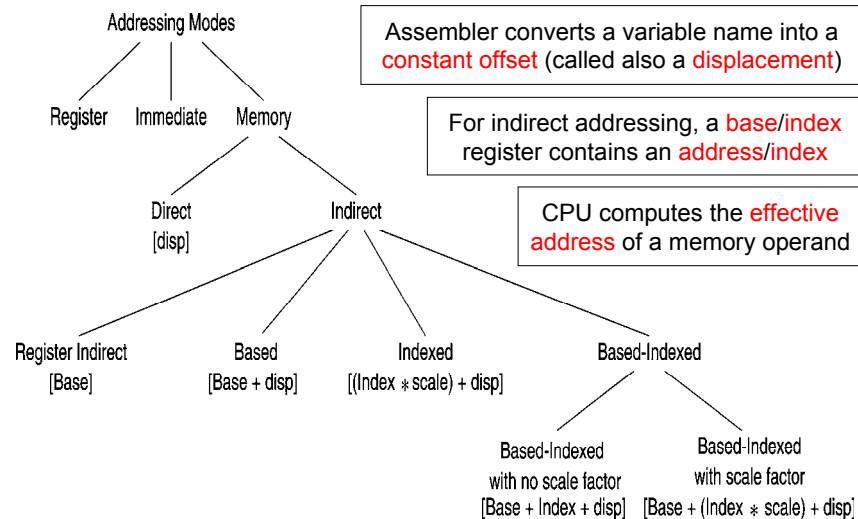
    lea eax, [ebx+esi*2]    ; mov eax, esi
                           ; add eax, eax
                           ; add eax, ebx
    
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 39

Summary of Addressing Modes



Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 40

Registers Used in 32-Bit Addressing

❖ 32-bit addressing modes use the following 32-bit registers

Base + (Index * Scale) + displacement

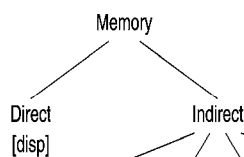
EAX	EAX	1	no displacement
EBX	EBX	2	8-bit displacement
ECX	ECX	4	32-bit displacement
EDX	EDX	8	
ESI	ESI		
EDI	EDI		
EBP	EBP		
ESP			

Only the index register can have a scale factor

ESP can be used as a base register, but not as an index

16-bit Memory Addressing

Old 16-bit addressing mode



Used with **real-address** mode

Only 16-bit registers are used

No Scale Factor

Only BX or BP can be the **base** register

Only SI or DI can be the **index** register

Displacement can be 0, 8, or 16 bits

Based-Indexed with no displacement

[BX + SI] [BP + SI]
[BX + DI] [BP + DI]

Based-Indexed with displacement

[BX + SI + disp]
[BX + DI + disp]
[BP + SI + disp]
[BP + DI + disp]

Default Segments

- ❖ When 32-bit register indirect addressing is used ...
 - ❖ Address in EAX, EBX, ECX, EDX, ESI, and EDI is relative to DS
 - ❖ Address in EBP and ESP is relative to SS
 - ❖ In flat-memory model, DS and SS are the same segment
 - Therefore, no need to worry about the default segment
- ❖ When 16-bit register indirect addressing is used ...
 - ❖ Address in BX, SI, or DI is relative to the data segment DS
 - ❖ Address in BP is relative to the stack segment SS
 - ❖ In real-address mode, DS and SS can be different segments
- ❖ We can override the default segment using segment prefix
 - ❖ `mov ax, ss:[bx]` ; address in bx is relative to stack segment
 - ❖ `mov ax, ds:[bp]` ; address in bp is relative to data segment

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 43

Next ...

- ❖ Operand Types
- ❖ Data Transfer Instructions
- ❖ Addition and Subtraction
- ❖ Addressing Modes
- ❖ **Jump and Loop Instructions**
- ❖ Copying a String
- ❖ Summing an Array of Integers

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 44

JMP Instruction

- ❖ JMP is an **unconditional jump** to a destination instruction
- ❖ Syntax: **JMP *destination***
- ❖ JMP causes the modification of the EIP register
 $EIP \leftarrow \text{destination address}$
- ❖ A **label** is used to identify the destination address
- ❖ Example:

```
top:
    . . .
    jmp top
```
- ❖ JMP provides an easy way to create a loop
 - ✧ Loop will continue endlessly unless we find a way to terminate it

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 45

LOOP Instruction

- ❖ The LOOP instruction creates a counting loop
- ❖ Syntax: **LOOP *destination***
- ❖ Logic: $ECX \leftarrow ECX - 1$
if $ECX \neq 0$, jump to *destination* label
- ❖ ECX register is used as a counter to count the iterations
- ❖ Example: calculate the sum of integers from 1 to 100

```
mov  eax, 0      ; sum = eax
mov  ecx, 100   ; count = ecx
L1:
add  eax, ecx   ; accumulate sum in eax
loop L1        ; decrement ecx until 0
```

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 46

Your turn . . .

What will be the final value of EAX?

Solution: 10

```
mov  eax,6
mov  ecx,4
L1:  inc  eax
     loop L1
```

How many times will the loop execute?

Solution: $2^{32} = 4,294,967,296$

What will be the final value of EAX?

Solution: same value 1

```
mov  eax,1
mov  ecx,0
L2:  dec  eax
     loop L2
```

Nested Loop

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

```
.DATA
count DWORD ?
.CODE
mov  ecx, 100    ; set outer loop count to 100
L1:  mov  count, ecx ; save outer loop count
     mov  ecx, 20  ; set inner loop count to 20
L2:  .
     .
     loop L2      ; repeat the inner loop
     mov  ecx, count ; restore outer loop count
     loop L1      ; repeat the outer loop
```


Next ...

- ❖ Operand Types
- ❖ Data Transfer Instructions
- ❖ Addition and Subtraction
- ❖ Addressing Modes
- ❖ Jump and Loop Instructions
- ❖ Copying a String
- ❖ Summing an Array of Integers

Copying a String

The following code copies a string from source to target

```
.DATA
    source BYTE "This is the source string",0
    target BYTE SIZEOF source DUP(0)
.CODE
main PROC
    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                 ; increment index
    loop L1                 ; loop for entire string
    exit
main ENDP
END main
```

↑
Good use of SIZEOF

↑
ESI is used to index source & target strings

Summing an Integer Array

This program calculates the sum of an array of 16-bit integers

```
.DATA
intarray WORD 100h,200h,300h,400h,500h,600h
.CODE
main PROC
    mov esi, OFFSET intarray    ; address of intarray
    mov ecx, LENGTHOF intarray ; loop counter
    mov ax, 0                   ; zero the accumulator
L1:
    add ax, [esi]               ; accumulate sum in ax
    add esi, 2                  ; point to next integer
    loop L1                    ; repeat until ecx = 0
    exit
main ENDP
END main
```

esi is used as a pointer
contains the address of an array element

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 51

Summing an Integer Array - cont'd

This program calculates the sum of an array of 32-bit integers

```
.DATA
intarray DWORD 10000h,20000h,30000h,40000h,50000h,60000h
.CODE
main PROC
    mov esi, 0                 ; index of intarray
    mov ecx, LENGTHOF intarray ; loop counter
    mov eax, 0                 ; zero the accumulator
L1:
    add eax, intarray[esi*4]   ; accumulate sum in eax
    inc esi                   ; increment index
    loop L1                   ; repeat until ecx = 0
    exit
main ENDP
END main
```

esi is used as a scaled index

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 52

PC-Relative Addressing

The following loop calculates the sum: 1 to 1000

Offset	Machine Code	Source Code
00000000	B8 00000000	mov eax, 0
00000005	B9 000003E8	mov ecx, 1000
0000000A		L1:
0000000A	03 C1	add eax, ecx
0000000C	E2 FC	loop L1
0000000E

When LOOP is assembled, the label L1 in LOOP is translated as FC which is equal to -4 (decimal). This causes the loop instruction to jump **4 bytes backwards** from the **offset of the next instruction**. Since the offset of the next instruction = 0000000E, adding -4 (FCh) causes a jump to location 0000000A. This jump is called **PC-relative**.

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 53

PC-Relative Addressing - cont'd

Assembler:

Calculates the difference (in bytes), called **PC-relative offset**, between the offset of the target label and the offset of the following instruction

Processor:

Adds the PC-relative offset to EIP when executing LOOP instruction

If the **PC-relative offset** is encoded in a single signed byte,

- what is the largest possible backward jump?
- what is the largest possible forward jump?

Answers: (a) -128 bytes and (b) +127 bytes

Basic Instructions & Addressing Modes

COE 205 – KFUPM

© Muhamed Mudawar – slide 54

Summary

❖ Data Transfer

- ❖ MOV, MOVSX, MOVZX, and XCHG instructions

❖ Arithmetic

- ❖ ADD, SUB, INC, DEC, NEG, ADC, SBB, STC, and CLC
- ❖ Carry, Overflow, Sign, Zero, Auxiliary and Parity flags

❖ Addressing Modes

- ❖ Register, immediate, direct, indirect, indexed, based-indexed
- ❖ Load Effective Address (LEA) instruction
- ❖ 32-bit and 16-bit addressing

❖ JMP and LOOP Instructions

- ❖ Traversing and summing arrays, copying strings
- ❖ PC-relative addressing