Basic Instructions & Assembly

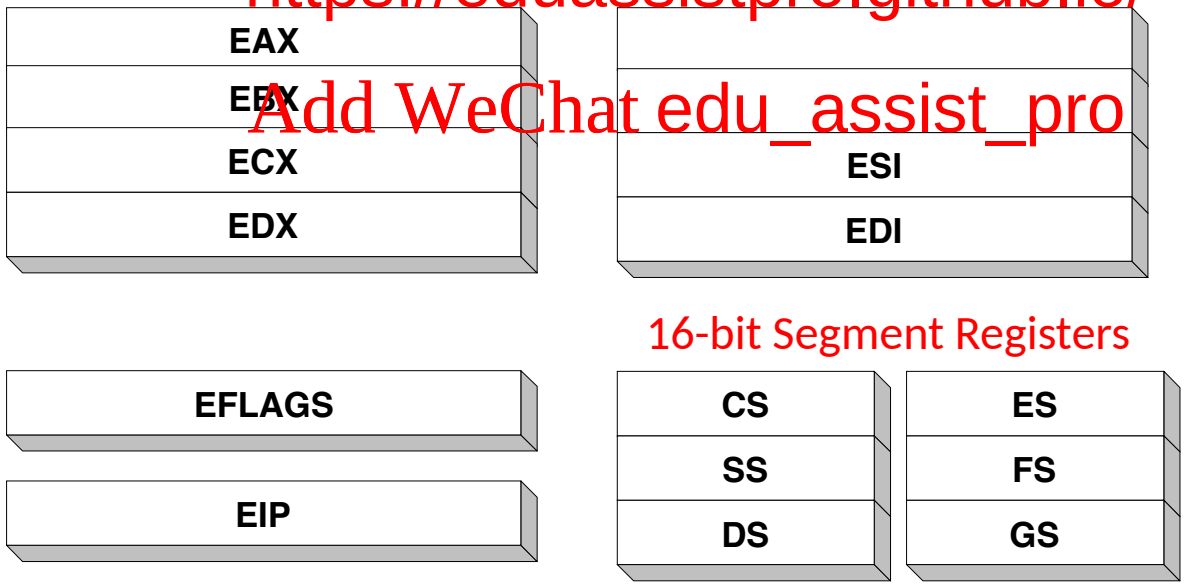
COMSC 260

Outline

- Operand Types
- Data Transfer Instructions
- Addition and
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

Basic Program Execution Registers

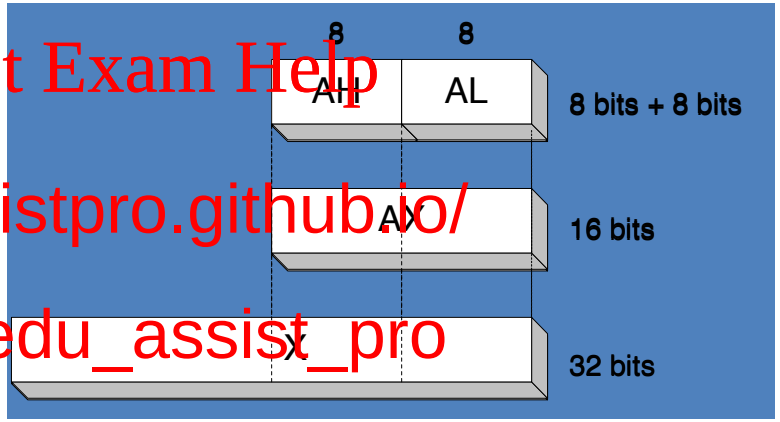
- Registers are high speed memory inside the CPU
  - Eight 32-bit general-purpose registers
  - Six 16-bit segment registers
  - Processor Status Flags (EFLAGS) and Instruction Pointer (EIP)



32 Bit Registers Description (review)

Accessing Parts of Registers

- **EAX, EBX, ECX, and EDX are 32-bit Extended registers**
  - Programmers can access their 16-bit and 8-bit parts
  - Lower 16-bit of EAX is named AX
  - AX is further divided into
    - AL = lower 8
    - AH = upper
- **ESI, EDI, EBP, ESP have only 16-bit names for lower half**



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

EFLAGS Register

Status Flags

Status of arithmetic and logical operations

Control and System flags

Control the CPU operation

Programs can set and clear individual bits in the EFLAGS register

Status Flags

Carry Flag

Set when unsigned arithmetic result is out of range

Overflow Flag

Set when signed arithmetic result is out of range

Sign Flag

Copy of sign bit, se

Zero Flag

Set when result is zero

Auxiliary Carry Flag

Set when there is a carry from bit 3 to bit 4

Parity Flag

Set when parity is even

Least-significant byte in result contains even number of 1s

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 n
- **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
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit se
imm	8-, 16-, o
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general-purpose register or memory byte
r/m16	16-bit operand which can be a 16-bit general-purpose register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	8-, 16-, or 32-bit memory operand

Next . . .

- Operand Types
- Data Transfer Instructions
- Addition and
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

MOV Instruction

Move source operand to destination

```
mov destination, source
```

Source and destination operands can vary

```
mov reg, reg
mov mem, re
mov reg, me
mov mem, imm
mov reg, imm
Real Mode:
mov r/m16, sreg
mov sreg, r/m16
```


ules

must be of same size

- No memory moves
- No immediate to segment moves
- Segment registers cannot be modified in protected mode
- Destination cannot be CS, EIP, IP

MOV Examples

```
.DATA
    count BYTE  100
    bVal  BYTE  20
    wVal  WORD  2
    dVal  DWORD 5
.CODE
    mov BL,  coun
    mov AX,  wVal
    mov count,AL     ; count = al
    mov EAX, dval    ; eax = dval

    ; 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
```

Copying Smaller Values to Larger ones

Positive Values

```
. DATA
    count WORD 1

    .CODE
    MOV ECX, 0
    MOV CX, count
```

Negative Values

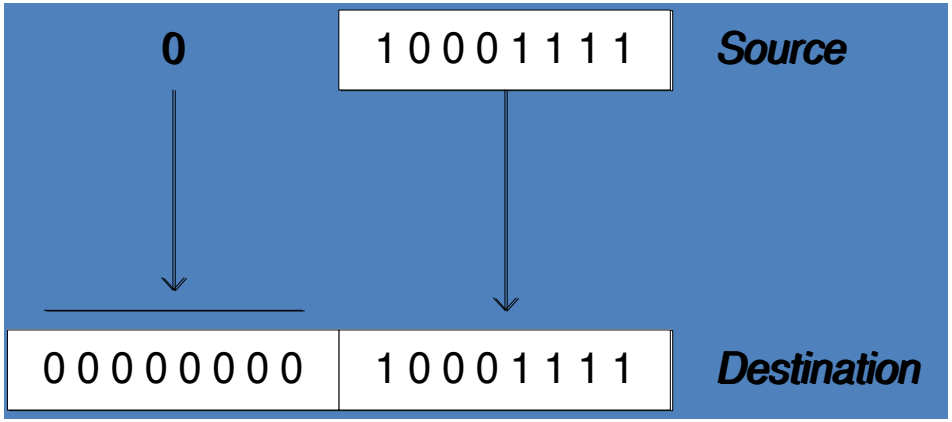
```
. DATA
    count SWORD -16      ; FFF0h

    0FFFFFFFFh
    ount
```

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,
movzx r16, r/m8
```

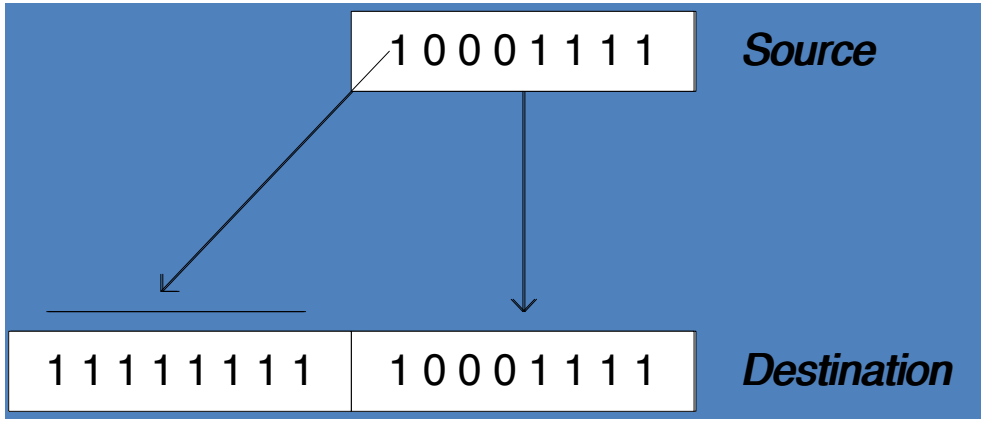

```
mov   bl, 8Fh
movzx ax, bl
```

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,
movsx r16,
```


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

LAHV and SAHV instructions

LAHV - Load Status Flags into AH
- Copies the low byte of the FLAGS register into AH

SAHV – Store A
- Copies AH int ister


```
.DATA
saveflags BYTE  ?
.CODE
LAHV                        ;load flags into AH
MOV saveflags , AH          ;save them in a variable
...
MOV AH, saveflags           ; load saved flags into AH
SAHV                        ; copy into FLAGS register
```

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 10000
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
```

Direct Memory Operands

- Variable names are references to locations in memory
- Direct Memory Operand:

  Named reference to a memory location
- Assembler compu ed variable


```
.DATA
var1 BYTE 10h
.CODE
mov AL, var1          ; AL = var1 = 10h
mov AL,[var1]         ; AL = var1 = 10h
```

Direct Memo

Alternate Format

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

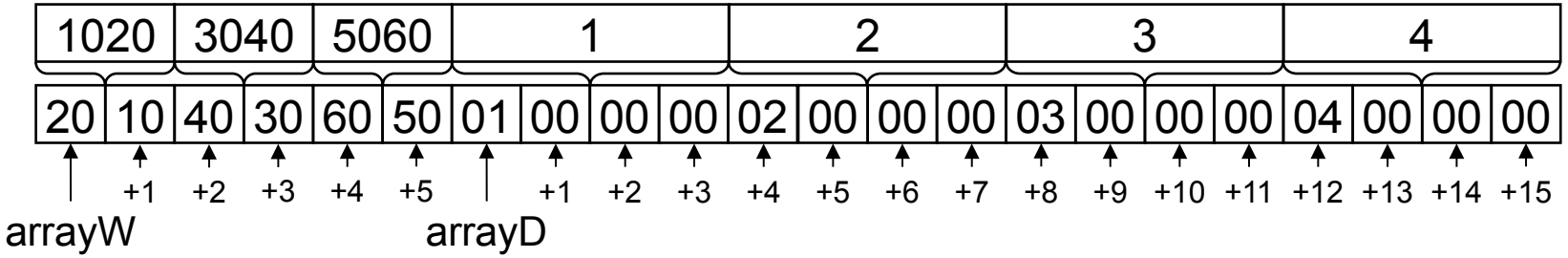
- Lets you access ve no name


```
.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`?

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
mov EAX,[arrayD-3
mov AX, [arrayW+9]    ; AX  = 02
mov AX, [arrayD+3]    ; Error: O          not same size
mov AX, [arrayW-2]    ; AX  = ? Out-of-range address
mov EAX,[arrayD+16]   ; EAX = ? MASM does not detect error
```

Your Turn . . .

Given the following definition of arrayD

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


Rearrange the th s: 3, 1, 2

Solution:


```
; Copy first array value i
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
```

Next . . .

- Operand Types
- Data Transfer Instructions
- Addition and
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

ADD and SUB Instructions

- ADD *destination*, *source*

  destination = destination + source

- SUB *destination*,

  destination = des

- Destination can be a *register* or a cation
- 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 8300h,1276h,0A456h
```


Solution: Accumu egister


```
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

1. Carry Flag: Set when unsigned arithme 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 ope ve
    - Two negative op ve
    - Overflow cannot occur when adding oper igns
- 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 ≠ 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 = carry out o
  - OF = (carry out of MSB) ⊕ (carry into MSB)

MSB = Most Significant Bit

⊕ = exclusive-OR operation

- Hardware does SUB by …
  - 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** = (carry out of the MSB) XOR (carry into the MSB)
  - **CF** = (INVERT) 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=0FFh    CF=1  OF=0  SF=1  ZF=0  AF=1  PF=1
mov AL,+127     ;
add AL,1        ; AL=80h     CF=0                    F=0  AF=1  PF=0
mov AL,26h
sub AL,95h      ; AL=91h     CF=1                    F=0  AF=0  PF=0
```


CF = 1 with the last subtract because if no carry, CF=0, inverted CF=1, if have to invert the CF (

	1	0	0	1	0	0	0	1	
−	0	0	1	0	0	1	1	0	26h (38)
	1	0	0	1	0	1	0	1	95h (-107)
	1	0	0	1	0	0	0	1	91h (-111)

	0	1	1	0	1	1	1	0	
+	0	0	1	0	0	1	1	0	26h (38)
	0	1	1	0	1	0	1	1	6Bh (107)
	1	0	0	1	0	0	0	1	91h (-111)

INC, DEC, and NEG Instructions

- **INC *destination***
  - *destination* = *destination* + 1
  - More compact (uses less space) than: ADD *destination*, 1
- **DEC *destination***
  - *destination* = *de*
  - More compact (uses less space) t *tination*, 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**

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 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=FFh      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   ; C=-128  CF=1  OF=1 SF=1 ZF=0 AF=0 PF=0
```

ADC and SBB Instruction

- Usually follows a normal add and sub instruction to deal with values twice as large as the size of the register. Used in 64 bit arithmetic

- ADC Instruction: Addition with Carry

  ```
  ADC destinatio
  destination =                    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 inte
- **Solution:**

```
add eax, ecx  ;add lower
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

ADC and SBB Instruction Example

Next . . .

- Operand Types
- Data Transfer Instructions
- Addition and
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

Addressing Modes

- Two Basic Questions
  - Where are the operands?
  - How memory addresses are computed?
- Intel IA-32 support modes
  - **Register** addressin
  - **Immediate** addressing: operand is stor ction 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

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 reg
- Immediate Addre
  - 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
```

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 (data label) to address memory directly
  - Assembler com
  - The variable off 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 Indire**
  - The memory a er
  - Brackets [ ] used to surround t olding 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, OFF                = array address
   mov EAX,[ESI                = [array] =
   10000h
   add ESI,4
   add EAX,[ESI]                              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 are addresses
  - The size of the memory operand is unclear 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 displacement (name ± constant) with an index register (all registers except ESP)

- Assembler converts displacement into a constant offset
- Constant offset is added to register to form an effective address

- **Syntax:** [*disp + i*


```
.data
   array DWORD 10000h,20000h
.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]
```

Index Scaling

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


```
.DATA
    arrayB BYTE  10h,20h,
    arrayW WORD  100h,200
    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
```

Based Addressing

- Syntax: [*Base* + *disp.*]
  - 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 Offse 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'
```

struct is a structure
isting of 3 fields:
a word, a double
word, and a byte

Based-Indexed Addressing

- Syntax: [*Base* + (*Index* * *Scale*) + *disp.*]
  - 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: h to start of array
  - Index register: selects an element > column index
  - Scaling factor: when array eleme 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                   bytes per row

.code
  mov ebx, 2*ROWSIZE                    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]
```

Summary of Addressing Modes

Assembler converts a variable name into a constant offset (called also a displacement)

For indirect addressing, a base/index register contains an address/index

computes the effective address of a memory operand

Registers Used in 32-Bit Addressing

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

Base + (Index * Scale) + displacement

EAX	EAX	1
EBX	EBX	2
ECX	ECX	4
EDX	EDX	8
ESI	ESI	
EDI	EDI	
EBP	EBP	
ESP		

no displacement

32-bit displac

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

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 indir
  - Address in BX, SI, or DI is relative segment DS
  - Address in BP is relative to the st 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

LEA Instruction

- LEA = Load Effective Address

```
LEA r32, mem     (Flat-Memory)
LEA r16, mem     (Real-Address Mode)
```

  - Calculate and l f a memory operand
  - Flat memory uses 32-bit effecti s
  - Real-address mode uses 16-bit dresses
- 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

LEA Examples

```
.data
   array WORD 1000 DUP(?)

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

   lea eax, arra                 ax, esi
      ; add eax,

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

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

Next . . .

- Operand Types
- Data Transfer Instructions
- Addition and
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

JMP Instruction

- JMP is an unconditional jump to a destination instruction
- Syntax: ***JMP destination***
- JMP causes the modification of the EIP register

  EIP ← destination
- A label is used to identify the destin s
- 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

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: **LOOP *destination***
- Logic: ECX ← ECX – 1
  if *tion* label
- ECX register is u the iterations
- Example: calculate the sum of i 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
```

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 ex

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
L1:
   mov count, ecx  ; save 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
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

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          egister
    mov  ecx, SIZEOF source                ter
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
```

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, OFFS intarray    ; address of intarray
    mov ecx, LENG
    mov ax,  0               ; 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

Summing an Integer Array – cont.

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, LENG
    mov eax, 0                  ; 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

Next . . .

- Operand Types
- Data Transfer Instructions
- Addition and
- Addressing M
- Jump and Loop Instruction
- Example Programs
  - Copying a String
  - Summing an Array of Integers
- PC-Relative Addressing

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`		
`0000000A`	`03 ; ecx`	
`0000000C`	`E2`	
`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.

PC-Relative Addressing – cont.

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 offs truction

If the PC-relative offset is encod gle signed byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

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

Summary

- Data Transfer
  - MOV, MOVSX, MOVZX, and XCHG instructions
- Arithmetic
  - ADD, SUB, INC, DEC, NEG, ADC, SBB, STC, and CLC
  - Carry, Overflow, ity flags
- Addressing Mod
  - Register, immediate, direct, indire ased-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