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
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 segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
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

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
```

Rules

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

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
```

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
```

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

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

Direct Memory Operand

Alternate Format

Direct-Offset Operands

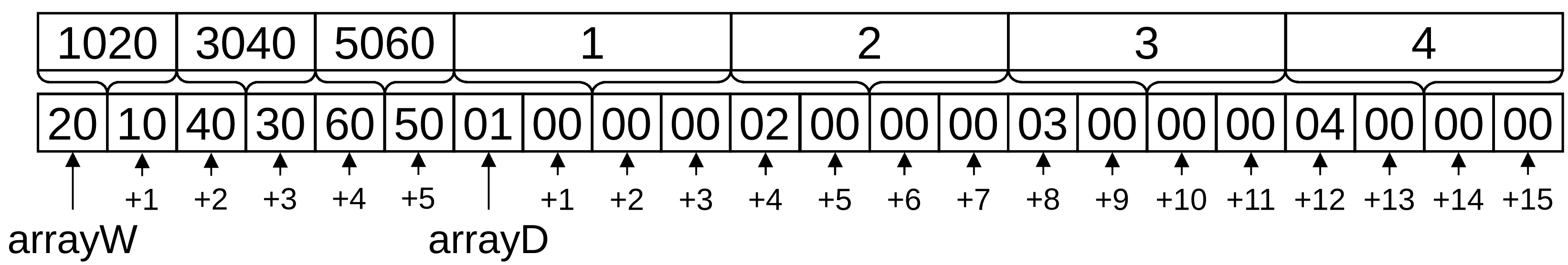
- Direct-Offset Operand: Constant offset is added to a named memory location to produce an effective address
  - Assembler computes the effective address
- Lets you access memory locations that have no name

```
.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`**?

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 = 00000002h
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
```



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
```

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
```

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

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
```

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

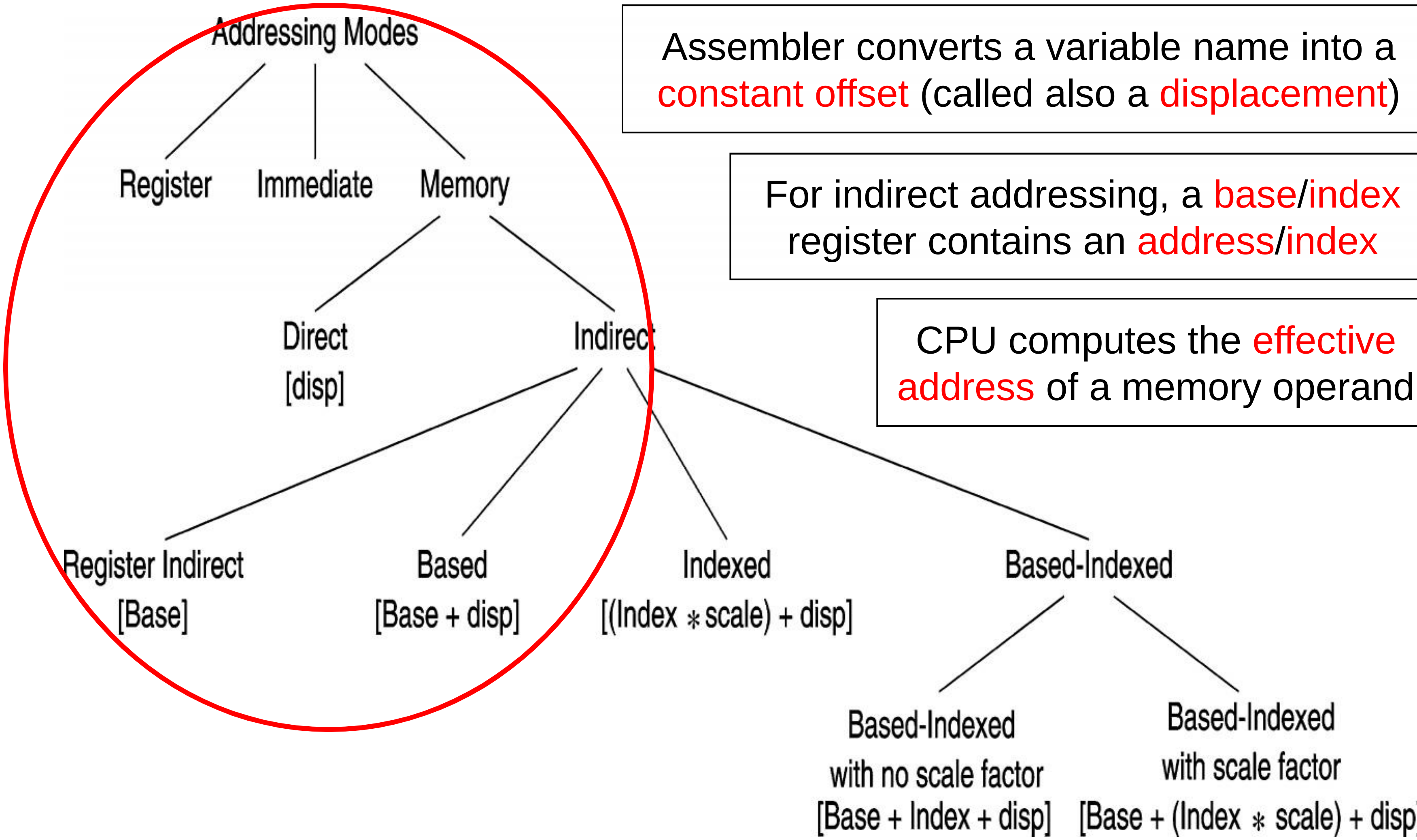
```
mov BYTE PTR [EBX], 100    ; BYTE operand in memory
add WORD PTR [ESI], 20     ; WORD operand in memory
inc DWORD PTR [EDI]        ; DWORD operand in memory
```

Indexed Addressing

- Combines a displacement (name±constant) with an index register
  - Assembler converts displacement into a constant offset
  - Constant offset is added to register to form an effective address
- Syntax: [*disp* + *index*] or *disp* [*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]
```

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

CPU computes the effective address of a memory operand

JMP Instruction

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

  EIP ← 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

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: **`LOOP destination`**
- Logic: ECX ← ECX – 1

  if ECX != 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
```

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
```

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

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