Arrays and Addressing Modes

Muhammad Bilal Amjad
www.facebook.com/bilal.amjad
bilalamjad78633@yahoo.com

Presentation Outline

- Operand Types
- Memory operands
- Addressing Modes
- 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
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

Next . . .

- Operand Types
- Memory operands
- Addition and Subtraction
- Addressing Modes
- Jump and Loop Instructions
- Copying a String
- Summing an Array of Integers

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

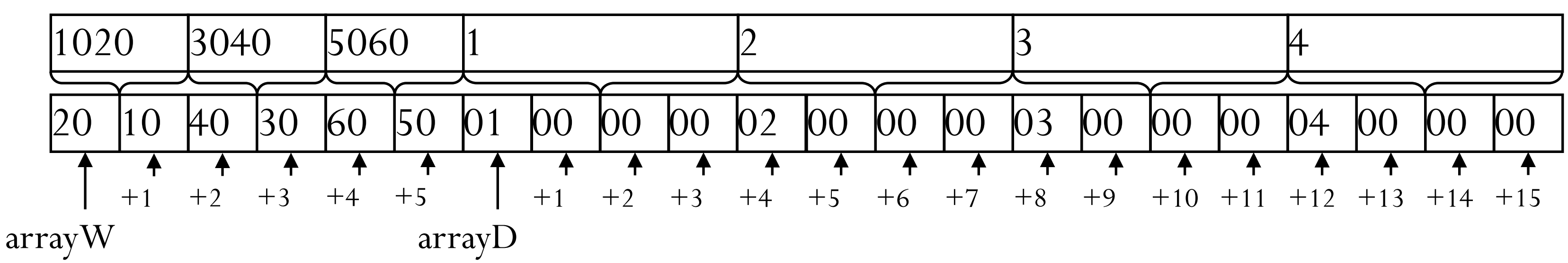
- Direct-Offset Operand: Constant offset is added to a named memory location to produce an effective address
  - Assembler computes the effective address
- Lets you access memory locations that have no name

```
.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
```



Next . . .

- Operand Types
- Memory operands
- Addressing Modes
- 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

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 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]
```

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
```

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]
```

LEA Instruction

- LEA = Load Effective Address

  LEA *r32*, *mem*   (Flat-Memory)

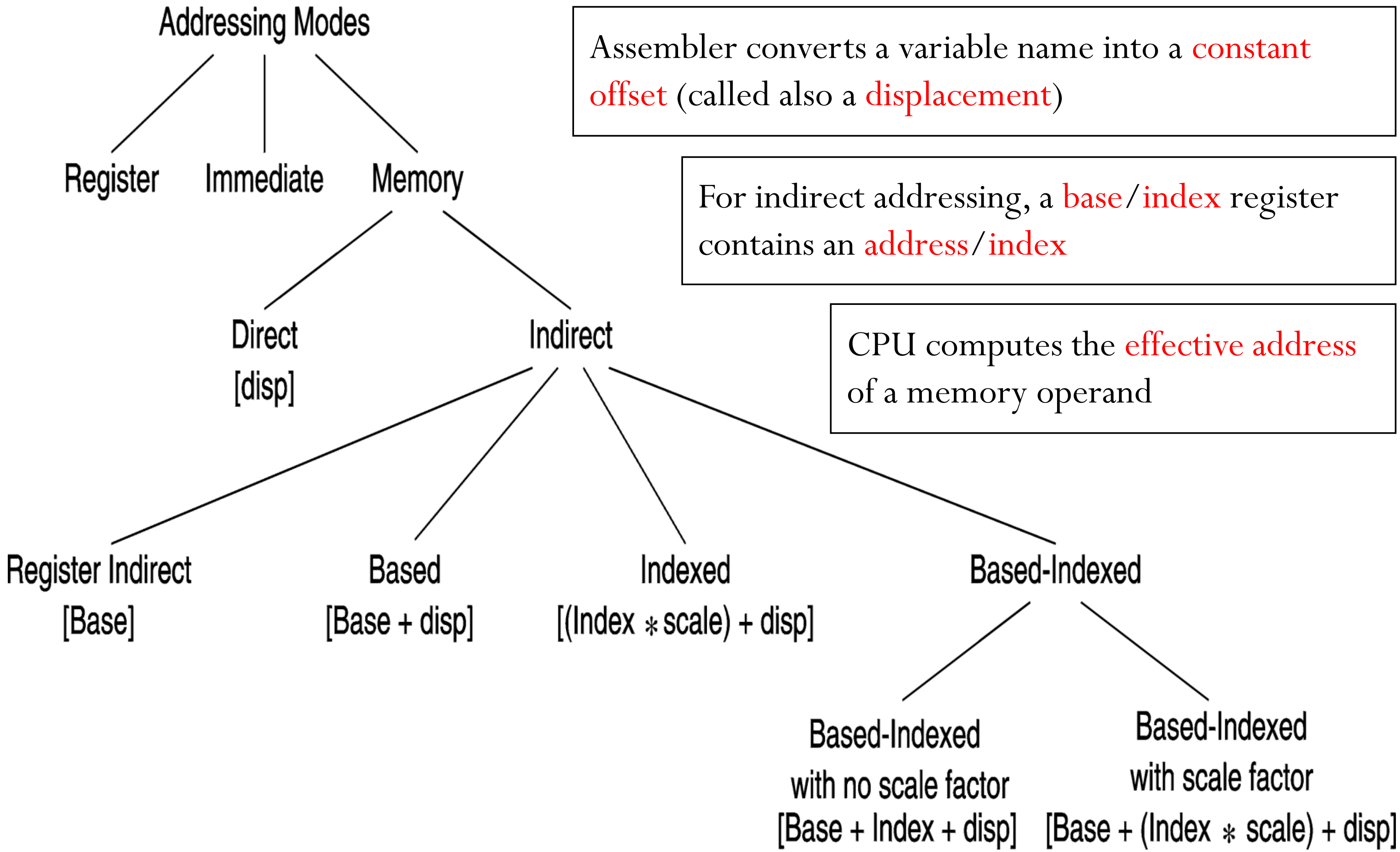
  LEA *r16*, *mem*   (Real-Address Mode)

  - 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

LEA Examples

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

.code                           ; Equivalent to . . .
  lea eax, array                ; 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
```

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

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

- Memory
  - Direct
    - [disp]
  - Indirect
    - Register Indirect
      - [BX] [BP] [SI] [DI]
    - Based
      - [BX + disp]
      - [BP + disp]
    - Indexed
      - [SI + disp]
      - [DI + disp]
    - Based-Indexed
      - 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]

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 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

Next . . .

- Operand Types
- Memory operands
- Addressing Modes
- 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

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

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.

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,

(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, MOVZX, and XCHG instructions
- 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