Arrays Access Methods/Modes

Direct Memory Addressing

- Used to address simple variables in memory
  - Variables are defined in the data section of the program
  - In syntax ,we use the variable name (label) to address memory directly
  - Assembler computes the offset of a variable from beginning of memory → so called Displacement only mode
  - The variable offset is specified directly as part of the instruction
- Example

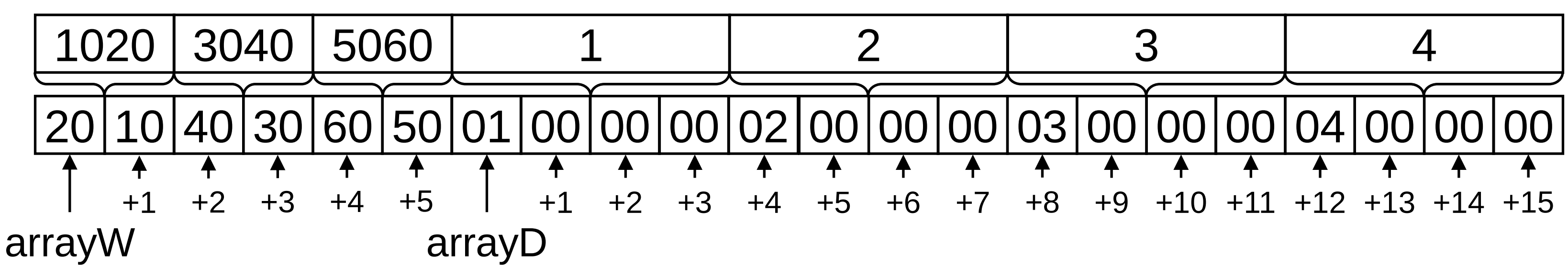
```
.data
    var1  DWORD   100
    var2  DWORD   200
    sum   DWORD   ?
.code
    mov eax, var1
    add eax, var2
    mov sum, eax
```

var1, *var2*, and *sum* are direct memory operands

Direct Memory Operands

Displacement (variable name) + constant Offset (can be added

```
.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
```



Mostly used Displacement + Indexed Addressing

Indexed Addressing (array access)

- Displacement only + index stored in register instead of adding constant offset (
- Mostly used register is ESI but can use EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP asl well
- **Syntax**: *displacement*[*index*], where
- Displacemet is array name + index is stored in ESI

```
.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]
```

Scaled Index

- ❖ Useful to index array elements of size 2, 4, and 8 bytes
  - ✧ **Syntax**: *Var-Name* [*index* * *scale*]

```
.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
```

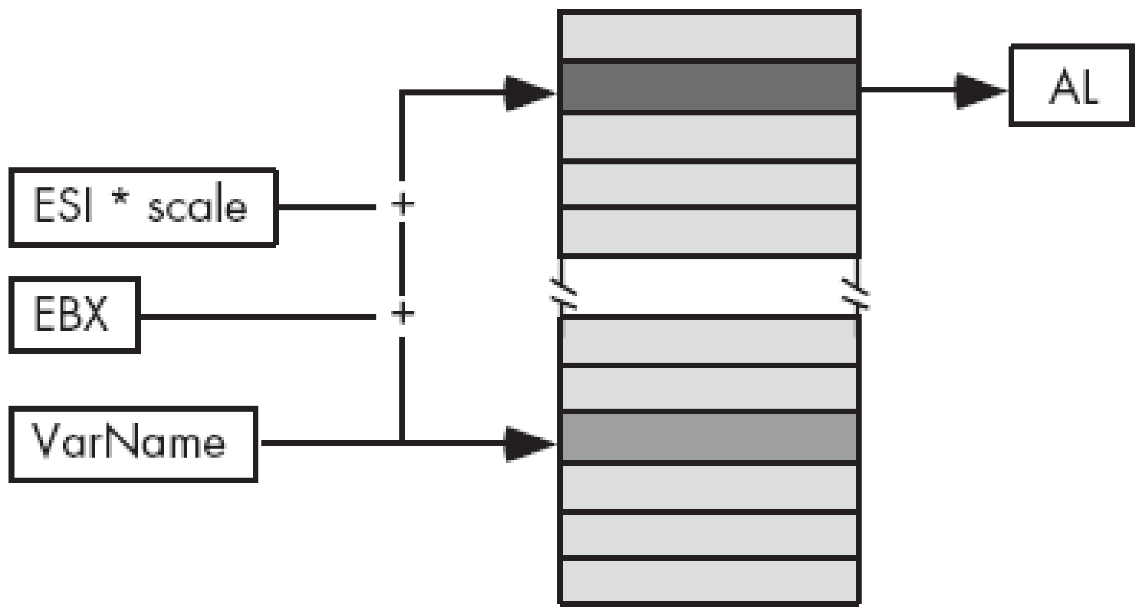
More Better: mov ax, arrayW[esi*TYPE arrayW]

Based [scaled] Indexed Addressing

- More General Form→ Addition of Base Register as well
- **Syntax**:

  VarName [Reg + Index-Register *Scale Factor]

- Used to access two-Dimensional Arrays



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]
```

Indirect Memory Addressing

- Register Indirect Addressing
  - The memory address is stored in a 32-bit register (EAX, EBX, ECX,EDX,ESI,EDI,EBP,ESP)
  - To access value, brackets [ ] used around the register holding the address
- 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

Note: EBX register is called Based register so this mode is also called Based Indirect Addressing,

Usually ESI is used to hold address.

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
```

Based Indirect Addressing with constant Offset

- EBX Register hold base address of some structure
- Syntax: [*Base* + *disp.*]
- 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

LEA Instruction

- LEA = Load Effective Address
  - Calculate and load the effective address of a memory operand
  - LEA Destination , Source
- 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

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

- Addressing Modes
  - Register
  - Immediate
  - Memory
    - Direct
      - Var Name → [disp]
    - Indirect
      - Register Indirect
        - [Base]
        - [ebx] or [esi] → offset array
      - Based
        - [Base + disp]
      - Indexed
        - [(Index * scale) + disp]
      - Based-Indexed
        - Based-Indexed with no scale factor
          - [Base + Index + disp]
        - Based-Indexed with scale factor
          - [Base + (Index * scale) + disp]

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

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

JMP Instruction

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

  EIP $\leftarrow$ *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
- 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

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`	`. . .`	`. . .`

Assembler: when LOOP is assembled, the label L1 in LOOP is translated as FC which is equal to –4 (decimal). [Jump will be backward]

It takes difference between the offset of the target label and the offset of the following instruction

CPU Adds the PC-relative offset (FC)to EIP (E) when executing LOOP instruction to find where to jump.

This jump is called PC-relative.

PC-Relative Addressing – cont'd

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

Covered up-till now

- Data representation (unsigned, signed, real numbers, characters, images)
- Data conversion (decimal to Binary/Hex and vice versa)
- Error Correction codes (CRC , Hamming)
- CPU IA-32 Architecture with registers detail
- Data Transfer instructions
  - MOV, MOVSX, MOVZX, and XCHG instructions
- Arithmetic instructions
  - 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
- JMP and LOOP Instructions
  - Traversing and summing arrays, copying strings
  - PC-relative addressing
- PC relative addressing

Thanks!