# Computer Organization and Assembly Language

Lecture 4 – Data Transfers, Addressing and Arithmetic

## Introducing Data Types in Assembler

- In higher-level languages (like Java and C++), the compiler is very strict in enforcing the rules regarding how data of different types are used.
- Assembly language does not enforce any of these rules; this requires that the programmer be more careful in declaring, moving and using data of different types.

# Operand Types

| Operand | Description |
|---|---|
| r8 | 8-bit general purpose register: AH, AL, BH, BL, etc. |
| r16 | 16-bit general purpose register: AX, BX, CX, DX |
| r32 | 16-bit general purpose register: EAX, EBX, etc. |
| 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 value |
| imm16 | 16-bit immediate value |
| imm32 | 32-bit immediate value |
| r/m8 | 8-bit operand which can be in a register or in memory |
| r/m16 | 16-bit operand which can be in register or memory |
| r/m32 | 32-bit operand which can be in register or memory |
| mem | an 8-, 16-, or 32-bit memory operand |

# Direct Memory Operands

- Imagine that your program contains the following in the data segment:

```
.data
var1     BYTE  10h
```

and that var1 is located at offset 10400h.You could write:

```
mov al, [00010400]
```

or

```
mov al, var1
```

- This is considered a direct memory operand because the value in the instruction is the actual offset.

## `mov` Instruction

- The ***mov*** instruction copies data from one location to another.
- The following formats are legal for moving data to or from general purpose registers:
  - **`mov`** *`reg, reg`*
  - **`mov`** *`mem, reg`*
  - **`mov`** *`reg, mem`*
- The following formats are legal for immediate operands
  - **`mov`** *`mem, immed`*
  - **`mov`** *`reg, immed`*
- The following format are legal for segment registers:
  - **`mov`** *`segreg, r/m16`* ; not CS
  - **`mov`** *`r/m16, segreg`*

## Moving Data From Memory to Memory

- Memory to memory moves cannot be done in a single instruction; it requires two instructions:

```
.data
var1 WORD ?
var2 WORD ?
… …
.code
     mov  ax, var1
     mov  var1, ax
```

# `mov` Instruction Examples

Examples of `mov` instructions

```
.data
   Count     BYTE   10
   Total     WORD   4126h
   Bigval    DWORD  12345678h

.code
   mov al, bl       ; 8-bit register to register
   mov bl, count    ; 8-bit memory to register
   mov count, 26    ; 8-bit immediate to memory
   mov bl, 1        ; 8-bit immediate to register
   mov dx, cx       ; 16-bit register to register
   mov bx, 8FE2h    ; 16-bit immediate to register
   mov eax, ebx     ; 32-bit register to register
   mov edx, bigVal  ; 32-bit memory to register
```

# Zero/Sign Extension of Integers

- Extending a 8- or 16-bit value into a 16- or 32-bit value is different for signed and unsigned values:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

if unsigned:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | ***( 214)*** |

if signed:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | ***(-42)*** |

## Copying Smaller Values into Larger Ones

- What happens if we write:

```
.data
count       WORD 1
.code
            mov         ecx, 0
            mov         cx, count; ECX is 0
```

## Copying Smaller Values into Larger Ones

- What happens if we write:

```
.data
signedVal   SWORD        -1
.code
            mov    ecx, 0
            mov    cx, SignedVal; ECX is
                              ; 0000FFF0h
                              ; (+65520)
```

- We could solve the problem by writing:

```
            mov    ecx, FFFFFFFFx
            mov    cx, signedVal ; ECX=FFFFFFF0h
                              ; (-16)
```

# MOVZX Instruction

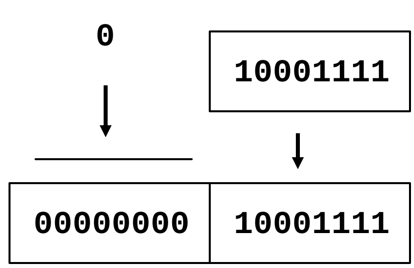
- The **`movzx`** (move with zero-extended) copies of the contents of source operand into a destination operand and zero-extends the operand to 16 or 32 bits.
- There are 3 formats:

  **`movzx`** *r32, r/m8*

  **`movzx`** *r32, r/m16*

  **`movzx`** *r16, r/m8*

# What **`movzx`** Does



## Examples Of `movzx`

- Register to register

```
mov      bx, 0A69Bh
movzx    eax, bx     ;    EAX = 0000A69Bh
movzx    edx, bl     ;    EDX = 0000009Bh
movzx    cx, bl      ;    CX = 009Bh
```

- Memory to register

```
.data
  byte1    BYTE  9Bh
  word1    WORD  0A69Bh
.code
  movzx    eax, word1  ;    EAX = 0000A69Bh
  movzx    edx, byte1  ;    EDX = 0000009Bh
  movzz    cx, byte1   ;    CX = 009Bh
```

## MOVSX Instruction

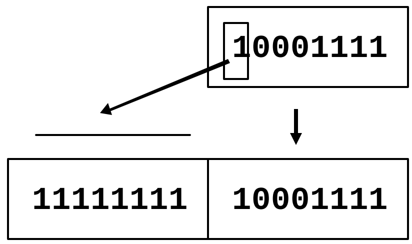
- The **`movsx`** (move with sign-extended) copies of the contents of source operand into a destination operand and sign-extends the operand to 16 or 32 bits.
- **`movsx`** is only used with signed integers.
- There are 3 formats:

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

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

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

# What **movsx** Does




# Examples Of **movsx**

- Register to register

```
mov       bx, 0A69Bh
movzx     eax, bx      ;     EAX = FFFFA69Bh
movzx     edx, bl      ;     EDX = FFFFFF9Bh
movzx     cx, bl       ;     CX = FF9Bh
```

# LAHF and SAHF Instructions

- LAHF (***L***oad into ***AH F***lags) copies the status flags in the low byte of EFLAGS into the AH register.
- The flags copies are Sign, Zero, Auxiliary Carry, Parity, and Carry.

```
.data
saveflags   BYTE  ?
.code
      lahf                      ; load flags into AH
      mov   saveflags, ah       ; save flags in memory
```

- SAHF (Save AH into Flags) copies the AH register values in the low byte of EFLAGS

```
mov ah, saveflags
sahf
```

# The **xchg** Instruction

- The ***xchg*** (exchange) instruction exhanges the contents of two memory locations.
- The syntax is:

```
xchg     reg, reg
xchg     reg, mem
xchg     mem, reg
```

- This does not require the use of a third location to swap values, making it very useful.

# The **xchg** Instruction - Examples

- Register-register or Register-Memory exchanges

```
xchg     ax, bx      ; exchange 16-bit regs
xchg     ah, al      ; exchange 8-bit regs
xchg     var1, bx    ; exchange 16-bit
                     ; memory operand with BX
xchg     eax, ebx    ;exchange 32-bit regs
```

- Memory-Memory exchange

```
mov      ax, value1  ; load the AX register
xchg     value2, ax  ; exchange AX and value2
mov      value1, ax  ; return AX to value
```

# Direct-Offset Operands

- A displacement can be added to the name of a memory operand, allowing the program to access data without their own memory labels:
- Examples:

```
arrayB   BYTE  10h, 20h
arrayW   WORD  100h, 200h
arrayD   DWORD 10000h, 20000h

mov      al, arrayB        ; AL = 10h
mov      al, arrayB+1      ; AL = 20h
mov      ax, arrayW;       ; AX = 100h
mov      ax, arrayW+2      ; AX = 200h
mov      eax, arrayD       ; EAX = 10000h
mov      eax, arrayD+4     ; EAX = 20000h
```

# Example – `moves.asm`

```
TITLE Data Transfer Examples  (Moves.asm)

INCLUDE     Irvine32.inc
.data
val1        WORD  1000h
val2        WORD  2000h
arrayB      BYTE  10h, 20h, 30h, 40h, 50h
arrayW      WORD  100h, 200h, 300h
arrayD      DWORD 10000h, 20000h

.code
main  PROC
```

```
; MOVZX
  mov       bx, 0A69Bh         ; Initialize BX reg
  movzx     eax, bx            ; EAX = 0000A69Bh
  movzx     edx, bl            ; EDX = 0000009Bh
  movzx     cx, bl             ; CX = 009Bh
  call      DumpRegs

; MOVSX
  mov       bx, 0A69Bh         ; Initialize BX reg
  movsx     eax, bx            ; EAX = FFFFA69Bh
  movsx     edx, bl            ; EDX = FFFFFF9Bh
  movsx     cx, bl             ; CX = FF9Bh
  call      DumpRegs
```

```
; Memory-to-memory exchange
  mov      ax, val1    ; AX = 1000h
  xchg     ax, val2    ; AX = 2000h,val2 = 1000h
  mov      val1, ax    ; val1 = 2000h
  call     DumpRegs

; Direct-offset Addressing (byte array)
  mov      al, arrayB        ; AL = 10h
  mov      al, [arrayB+1]    ; AL = 20h
  mov      al, [arrayB+2]    ; AL = 30h
  call     DumpRegs
```

```
; Direct-offset Addressing (word array)
      mov  ax, arrayW        ; AX = 100h
      mov  ax, [arrayW+2]    ; AX = 200h
      call DumpRegs

; Direct-offset Addressing (doubleword array)
      mov  eax, arrayD       ; EAX = 100h
      mov  eax, [arrayD+4]   ; EAX = 200h
      call DumpRegs


      exit
main  ENDP
      END  main
```

# Arithmetic Instructions

Assembly language include many instructions to perform basic arithmetic. They include:

- inc
- dec
- add
- sub

# **`inc`** and **`dec`** Instructions

- The ***inc*** and ***dec*** instructions have the format:

```
inc     reg/mem     ; add 1 to destination's
                    ; contents
dec     reg/mem     ; subtract 1 to
                    ; destination's contents
```

- The operand can be either a register or memory operand.
- All status flags (except Carry) are affected.

# `inc` and `dec` - Examples

- Simple examples

```
inc     al    ; increment 8-bit register
dec     bx    ; decrement 16-bit register
inc     eax   ; increment 32-bit register
inc     val1  ; increment memory operand
```

- Another example

```
.data
myWord      WORD  1000h
.code
      inc   myWord      ; 1001h
      mov   bx, myWord
      dec   bx          ; 1000h
```

# `add` Instruction

- ***add*** adds a *source* operand to the *destination* operand of the **same size**.
- Format:

  `add` *destination*, *source*
- *Source* is unchanged; *destination* stores the sum. All the status flags are affected.
- The sizes must match and only one can be a memory location.

## **add** Instruction - Examples

- Simple examples

```
add     cl, al       ; add 8-bit register to register
add     eax, edx     ; add 32-bit register-to-register
add     bx, 1000h    ; add immediate value to 16-bit reg
add     var1, ax     ; add 16-bit register to memory
add     var1, 10     ; add immediate value to memory
```

- Numeric example

```
.data
var1  DWORD 10000h
var2  DWORD 20000h
.code
      mov   eax, var1
      add   eax, var2   ; 30000h
```

## **sub** Instruction

- ***sub*** subtracts a *source* operand from the *destination* operand of the **same size**.
- Format:

  **sub** *destination*, *source*
- *Source* is unchanged; *destination* stores the difference. All the status flags are affected.
- The sizes must match and only one can be a memory location.

## **sub** Instruction - Examples

- Simple examples

```
sub      12345h, eax ; 32-bit immediate from reg
sub      cl, al      ; 8-bit reg from reg
sub      var1, ax    ; 16-bit reg from memory
sub      dx, var1    ; 16-bit memory from reg
sub      var1, 10    ; immediate from memory
```

- Numeric example

```
.data
var1  DWORD 30000h
var2  DWORD 10000h
.code
      mov   eax, var1
      sub   eax, var2   ; 10000h
```

## Flags Affected by **add** and **sub**

- If **add** or **sub** generates a result of zero, ZF is set
- If **add** or **sub** generates a negative result, SF is set.
- Examples:

```
mov      ax, 10
sub      ax, 10      ; AX = 0, ZF = 1
mov      bx, 1
sub      bx, 2       ; BX = FFFF, SF = 1
```

- **inc** and **dec** affect ZF but not CF.

```
mov      bl, 4Fh
add      bl, 0B1h    ; BF = 00, ZF = 1, CF = 1
mov      ax, 0FFFFh
inc      ax          ; ZF = 1 (CF unchanged)
```

## Flags Affected by **add** and **sub** (continued)

- The Carry flag is useful when performing unsigned arithmetic

```
mov     ax, 0FFh
add     al, 1       ; AL = 00, CF = 1
```

- This should have been a 16-bit operation:

```
mov     ax, 0FFh
add     ax, 1       ; AX = 0100, CF = 0
```

- A similar situation happens when subtracting a larger unsigned value from a smaller one:

```
mov     al, 1
sub     al, 2       ; AL = FF, CF = 1
```

## Flags Affected by **add** and **sub** (continued)

- The Overflow flag is useful when performing signed arithmetic:

```
mov     al, +126
add     al, 2       ; AL = 80h, OF = 1
```

| | |
|---|---|
| 126 | 01111110 |
| +2 | + 00000010 |
| -128 | 1 00000000 |

```
mov     al, -128
sub     al, 2       ; AL = 7Eh, OF = 1
```

| | |
|---|---|
| -128 | 10000000 |
| - -2 | -00000010 |
| -130 | 0 11111110 |

# Implementing Arithmetic Expressions

- Imagine we are implementing the statement

```
  Rval = -Xval + (Yval – Zval)
.data
Rval        SDWORD      ?
Xval        SDWORD      26
Yval        SDWORD      30
Zval        SDWORD      40
.code
; first term: -Xval
      mov   eax. Xval
      neg   eax         ; EAX = -26
```

# Implementing Arithmetic Expressions (continued)

```
; second term: (Yval – Zval)
      mov   ebx, Yval
      sub   ebx, Zval   ; EBX = -10

; add the terms and store
      add   eax, ebx
      mov   Rval, eax   ; Rval = -36
```

# Example Program: **AddSum3.asm**

```
TITLE  Addition and Subtraction (AddSum3.as)

INCLUDE       Irvine32.inc
.data
Rval   SDWORD ?
Xval   SDWORD 26
Yval   SDWORD 30
Zval   SDWORD 40

.code
main   PROC
   ; INC and DEC
   mov ax, 1000h
   inc ax     ; 1001h
   dec ax     ; 1000h
   call DumpRegs
```

```
   ; Expression: Rval = -Xval + (Yval - Zval)
   mov eax, Xval
   neg eax         ; EAX = -26
   mov ebx, Yval
   sub ebx, Zval   ; EBX = -10
   add eax, ebx
   mov Rval, eax   ; Rval = -36
   call      DumpRegs

   ; Zero flag example
   mov cx, 1
   sub cx, 1       ; ZF = 1
   mov ax, 0FFFFh
   inc ax          ; ZF = 1
   call      DumpRegs
```

```
; Sign flag example
mov cx, 0
sub cx, 1       ; SF = 1
mov ax, 7FFFh
add ax, 2       ; ZF = 1
call      DumpRegs

; Carry flag example
mov al, 0FFh
add al, 1       ; CF = 1, AL = 00
call      DumpRegs
```

```
      ; Overflow flag example
      mov   al, +127
      add   al, 1       ; OF = 1
      mov   al, -128
      sub   al, 1       ; OF = 1
      call  DumpRegs

      exit
main  ENDP
      END main
```

## **NEG** Instruction

- The NEG instruction reverses the sign of an operand (in two's complement form):

```
NEG     reg
NEG     mem
```

- **NEG** affects the same flags as **AND**, **OR** and **XOR.**
- It is important to check the Overflow flag after **NEG** in case you have negated –128 to +128, producing an invalid answer.

```
mov     al, -128     ; AL = 10000000b
neg     al           ; AL = 10000000b, OF = 1
```

- +127 can be negated without problem

```
mov     al, + 127    ; AL = 01111111b
neg     al           ; AL = 10000001b
```

## OFFSET Operator

- The OFFSET operator returns the number of bytes between the label and the beginning of its segment.
- In ***Real mode*** it produces a ***16-bit*** immediate value; therefore, the destination must be a 16-bit operand.
- In ***Protected mode*** it produces a ***32-bit*** immediate value; therefore, the destination must be a 32-bit operand.

# OFFSET Example in 16 Bits

```
mov   bx, offset count  ; BX points to count
```

*or*

```
.data
bList db    10h, 20h, 30h, 40h
wList dw    1000h, 2000h, 3000h
.code
mov   di, offset bList  ; DI = 0000
mov bx, offset bList+1  ; BX = 0001
mov si, offset wList+2  ; SI = 0006
```

# OFFSET Example in 32 Bits

```
.data
bVal        BYTE        ?
wVal        WORD        ?
dVal        DWORD       ?
dVal2       DWORD       ?
```

- IF bVal is located at offset 00404000h, we would get:

```
mov   esi, OFFSET bval        ; ESI = 00404000
mov   esi, OFFSET wVal        ; ESI = 00404001
mov   esi, OFFSET dVal        ; ESI = 00404003
mov   esi, OFFSET dVal2       ; ESI = 00404007
```

# `ALIGN` Directive

- ALIGN aligns the next variable on a byte, word, doubleword, or paragraph boundary.
- The syntax is

  `ALIGN` *bound*

  where bound is 1, 2 or 4.
- Example

```
bVal  BYTE  ?          ; 00404000
      ALIGN 2
wVal  WORD  ?          ; 00404002
bVal  BYTE  ?          ; 00404004
      ALIGN 4
dVal  DWORD ?          ; 00404008
DVal2 DWORD ?          ; 0040400C
```

# Data-Related Operators and Directives

- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

## PTR Operator

- PTR operator overrides the default size for a operand's address.
- It is useful when operand's size is not clear from the context:

```
inc [bx]
```

would produce an "operand must have size error message."We can fix it by writing

```
inc             byte ptr [bx]
```

- PTR is useful in overriding default sizes for an operand:

```
.data
val32    DWORD       12345678h
.code
mov      ax, word ptr val32      ; AX = 5678H
mov      dx, word ptr val32+2    ; DX = 1234H
```

## **PTR** Operators - An Example

```
.data
myDouble    DWORD 12345678h
.code
      mov   ax, myDouble            ; ERROR
      mov   ax, word ptr MyDouble   ; WORKS!
```

| Address | Byte |
|---|---|
| 0000 | 78 |
| 0001 | 56 |
| 0002 | 34 |
| 0003 | 12 |

| | |
|---|---|
| myDouble | 12345678 |
| word ptr MyDouble | 5678 |
| byte ptr MyDouble | 78 |
| byte ptr [myDouble+1] | 56 |
| word ptr [myDouble+2] | 1234 |
| byte ptr [myDouble+2] | 34 |

# TYPE Operator

- The TYPE operator returns the size (in bytes) of a single element of a variable.
  - For variables, it is 1, 2 or 4 for bytes, words and doublewords respectively.
  - For near labels, it is FFFFh; for far labels FFFEh.
- Example

```
.data
var1        BYTE        ? ; TYPE var1 = 1
var2        WORD        ? ; TYPE var2 = 2
var3        DWORD       ? ; TYPE var3 = 4
var4        QWORD       ? ; TYPE var4 = 8
```

# TYPE Operator - An Example

```
.data
var1     BYTE  20h
var2     WORD  1000h
var3     DWORD ?
var4     BYTE  10, 20, 30, 40, 50
msg      BYTE  'File not found', 0
.code
L1:mov ax, type var1       ; AX = 0001
   mov ax, type var2       ; AX = 0002
   mov ax, type var3       ; AX = 0004
   mov ax, type var4       ; AX = 0001
   mov ax, type msg        ; AX = 0001
   mov ax, type L1         ; AX = FFFF
```

# LENGTHOF Operator

- The **LENGTHOF** operator counts the number of individual elements in a variable that has been defined using DUP.
- Example

```
.data
val1       WORD   1000h
val2       SWORD  10, 20, 30
array      WORD   32 dup(0)
array2     WORD   5 dup(3dup(0))
message    BYTE   'File not found', 0
.code
    mov    ax, LENGTHOF val1          ; AX = 1
    mov    ax, LENGTHOF val2          ; AX = 1
    mov    ax, LENGTHOF array         ; AX = 32
    mov    ax, LENGTHOF array2        ; AX = 5
    mov    ax, LENGTHOF message       ; AX = 1
```

# SIZEOF Operator

- The **SIZEOF** operator returns the number of bytes an array takes up.
- It is equivalent to multiplying **LENGTHOF** by **TYPE**.
- Example

```
intArray  WORD 32 DUP(0) ; SIZEOF = 64
```

# **LABEL** Directive

- **LABEL** operator inserts a label without allocating storage. It points to the same address as the variable declared below it.
- Example

```
.data
val16    LABEL word
val32    DWORD 12345678h
.code
mov      ax,   val16       ; AX  = 5678h
mov      dx,   val32+2     ; DX = 1234h
```

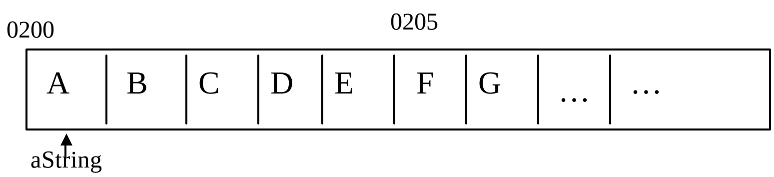
# Indirect Operands

- An indirect operand is a register containing the offset for data in a memory location.
  - The register points to a label by placing its offset in that register
  - This is very convenient when working with arrays; it is just a matter of incrementing the address so that it points to the next array element.
  - The ESI, EDI, EBX, EBP, SI, DI, BX and BP registers can be used for indirect operands as well as the 32-bit general purpose registers (with a restriction on the ESP).

# Indirect Operands: A Real Mode Example

- We create a string in memory at offset 0200 and set the BX to the string's offset; we can process any element in the string by adding to the offset:
- `.data`

```
……
aString  BYTE  "ABCDEFG"
.code
mov      bx, offset aString      ; BX = 0200
add      bx, 5                   ; BX = 0205
mov      dl, [bx]                ; DL = 'F'
```




# Indirect Operands: A Protected Mode Example

```
.data
val1  BYTE  10h
.code
      mov   esi   OFFSET val1
      mov   al, [esi]         ; AL = 10h
      mov   [esi], bl   ; The variable to
                        ; which ESI points is
                        ; changed
      mov   esi, 0
      mov   ax, [esi]   ; General Protection
                        ; Error
      inc   [esi] ; Error - needs size
      inc   byte ptr [esi]    ; Works!
```

# Arrays

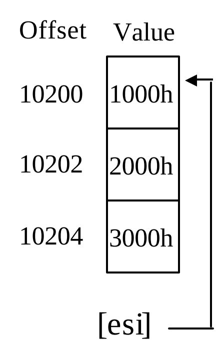
- Indirect arrays are useful when manipulating arrays:

```
.data
arrayB      BYTE        10h, 20h, 30h
.code
      mov   esi, OFFSET arrayB
      mov   al, [esi]         ; AL = 10h
      inc   esi
      mov   al, [esi]         ; AL = 20h
      inc   esi
      mov   al, [esi]         ; AL = 30h
```

# Arrays of Words

- If we use an array of 16-bit integers, we add 2 to ESI to address each subsequent array element:

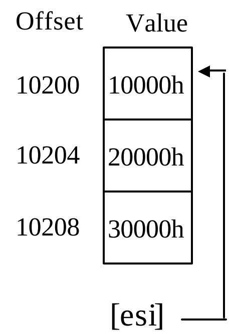
```
.data
arrayW      WORD  1000h, 2000h, 3000h
.code
      mov   esi, OFFSET arrayW
      mov   ax, [esi]   ; AX = 1000h
      add   esi, 2
      mov   ax, [esi]   ; AX = 2000h
      add   esi, 2
      mov   ax, [esi]   ; AX = 3000h
```



# Arrays of Doublewords

- If we use an array of 32-bit integers, we add 4 to ESI to address each subsequent array element:

```
.data
arrayD  DWORD   10000h, 20000h, 30000h
.code
      mov   esi, OFFSET arrayD
      mov   eax, [esi]  ; first #
      add   esi, 4
      mov   eax, [esi]  ; second #
      add   esi, 4
      mov   eax, [esi]  ; third #
```




# Indexed Operands

- An indexed operand adds a constant to a register to generate an effective address.
- Any of the 32-bit general purpose register may be used as an index registers.
- There are two forms that are legal:

```
constant[reg]
[constant+reg]
```

- In both cases, we are combining the constant offset of a variable label with the contents of a register.

## Indexed Operands – An Example

```
.data
arrayB      BYTE  10h, 20h, 30h
arrayW      WORD  1000h, 2000h, 3000h
.code
      mov   esi, 0
      mov   al, [arrayB+esi]  ; AL = 10h

      mov   esi, OFFSET arrayW
      mov   ax, [esi]         ; AX = 1000h
      mov   ax, [esi+2]       ; AX = 2000h
      mov   ax, [esi+4]       ; AX = 3000h
```

## Indexed Operands Using 16-Bit Registers

- The 16-bit registers can be used as indexed operand in real mode; however, you are limited to the SI, DI BP and BX registers:

```
mov     al, arrayB[si]
mov     ax, arrayW[di]
mov     eax, ArrayD[bx]
```

- As with indirect operand, avoid using the BP except when addressing data on the stack.

# Pointers

- A variable that contains the address of another variable is called a ***pointer*** (because it ***points*** to the variable).
- In Intel assembler, there are two basic types of pointers:
  - `NEAR` pointers, an offset from the beginning of the data segment (in **real** mode, a **16-bit** offset; in **protected** mode, a **32-bit** offset).
  - `FAR` pointers, a segment-offset address (in **real** mode, a **32-bit** address; in **protected** mode, a **48-bit** address).

## Initializing Pointers

- Near pointers in protected are stored in doubleword variables:

```
arrayB  BYTE      10h, 20h, 30h, 40h
arrayW  WORD      1000h, 2000h, 3000h
ptrB    DWORD     arrayB
ptrW    DWORD     arrayW
```

- The OFFSET operator can also be used in initializing pointers to make the relationship clearer:

```
ptrB    DWORD     OFFSET arrayB
ptrW    DWORD     OFFSET arrayW
```

## Using the TYPEDEF Operator

- The TYPEDEF operator lets you create user-defined types that can be used in the same way as the built-in type:

```
PBYTE    TYPEDEF     PTR BYTE
.data
arrayB   BYTE  10h, 20h, 30h, 40
ptr1     PBYTE ?            ; uninitialized
ptr2     PBYTE arrayB       ; points to an
                            ;     array
```

## Pointers: An Example

```
TITLE Pointers    (Pointers.asm)

INCLUDE     Irvine32.inc

; Create user-defined types
PBYTE TYPEDEF     PTR BYTE    ; points to bytes
PWORD TYPEDEF     PTR WORD    ; points to words
PDWORD      TYPEDEF     PTR DWORD
                        ; points to doublewords

.data
arrayB      BYTE  10h, 20h, 30h
arrayW      WORD  1, 2, 3
arrayD      DWORD 4, 5, 6
```

```
; create some pointer variables
ptr1   PBYTE  arrayB
ptr2   PWORD  arrayW
ptr3   PDWORD arrayD

.code
main   PROC
; Use the pointers to access data
   mov esi, ptr1
   mov al, [esi]     ; 10h
   mov esi, ptr2
   mov ax, [esi]     ; 1
   mov esi, ptr3
   mov eax, [esi]    ; 4h
   exit
main   endp
   END main
```

# Transfer of Control

- A transfer of control is way of altering the order in which instructions are executed.
- The two basic ways are:
  - ***Unconditional transfer*** – the program branches to a statement elsewhere in the program
  - ***Conditional transfer*** – the program branches to a statement elsewhere in the program ***IF*** some condition is true.

# JMP Instruction

- The JMP statement causes an unconditional transfer to the target address within the same code segment.
- The syntax is:

  **`JMP`** *targetLabel*

  where the targetLabel is the offset of an instruction elsewhere in the program.
- Example:

```
top:
   … …
     jmp top; infinite loop
```

# `LOOP` Instruction

- The LOOP instruction is used to end a block of statements that will be performed a predetermined number of times, with the number of times stored in the ECX (or CX) register.
- The syntax is:

  **`LOOP`** *destination*

  where destination is the label of the statement to which it jumps if the (E)CX register is nonzero.
- Because the (E)CX register controls the loop, it is extremely unwise to change it during the loop.

## Using the ECX Register During Loops

- If it is necessary to use the (ECX) register during loops, it is important to restore its value before the **LOOP** instruction:

```
.data
count    DWORD ?
.code
  mov    ecx, 100
  … …
  mov    count, ecx   ; save the count
  … …
  mov    ecx, 20      ; modify ECX
  … …
  mov    ecx, count   ; restore Loop count
  loop   top
```

## Nested Loops

- In writing nested loops, it is important to save the outer loop's counter:

```
.data
count DWORD ?
.code
      mov   ecx, 100    ; set outer loop’s count
L1:   mov   count, ecx  ; save outer loop count
      mov   ecx, 20     ; set inner loop count
L2:   … …
      loop  L2          ; repeat inner loop
      mov   ecx, count  ; restore outer loop count
      loop  L1
```

## Summing An Integer Array

```
TITLE Summing An Array  (SumArray.asm)

INCLUDE     Irvine32.inc
.data
intarray    WORD  100h, 200h, 300h, 400h

.code
main  PROC
  mov edi, OFFSET intarray    ; address of
  intarray
  mov ecx, LENGTHOF intarray; ; loop counter
  mov ax, 0
```

```
L1:
  add ax, [edi]         ; add an integer
  add edi, TYPE intarray; point to next integer
  loop      L1          ; repeat ECX = 0
  call DumpRegs
  exit
main  endp
  end main
```

# Copying A String

```
TITLE Copying A String  (CopyStr.asm)

INCLUDE     Irvine32.inc
.data
source      BYTE  "This is the source string",0
target      BYTE  SIZEOF      source DUP(0), 0

.code
main  PROC
  mov esi, 0                  ; index register
  mov ecx, SIZEOF source      ; loop counter
```

```
L1:
  mov al, source[esi]   ; get a char. from source
  mov target[esi], al   ; store it in the target
  inc esi         ; move it to next character
  loop      L1          ; repeat for whole string

  exit
main  endp
  end main
```