4. Data Transfers, Addressing & Arithmetic

Thursday, August 22, 2024 9:27 AM

* Three basic types of operands
  * Immediate operand – numeric & charactic literal
  * Register Operand – named CPU register
  * Memory operand – memory location

```
.data
    var1  byte  10h

.code
    mov   al, var1
```

machine language → A0 (operand code) 00010400 (32-bit hex. address to var1) — assembled

MOV Instruction

* Operands must be same size
* both operands cannot be memory
* Instruction Pointer register (IP, EIP, RIP) cannot be destination operand

```
mov   reg, reg
      mem, reg
      reg, mem
      mem, imm
      reg, imm
```

MOVZX Instruction

* move with zero-extend copies source to destination and extends to 16 or 32-bits
* used with unsigned integers (positives)

```
movzx    reg32, reg/mem8
         reg32, reg/mem16
         reg16, reg/mem8
```

* add leading zeros if necessary
* operands don't have to be same size

```
.data
    var1 word 1234h

.code
    movzx eax, var1
```

(eax → 4-bytes, var1 → 2-bytes)

```
  FFFFh   (word)
+ 0001h   (word)
---------
 10000h   (dword)
```

EAX: | 0000 | 1234 |

* MOVSX Instruction

* move with sign-extend copies source to destination & sign extends to 16 or 32-bits
* used with signed integers

```
movsx    reg32, reg/mem8
         reg32, reg/mem16
         reg16, reg/mem8
```

* Highest bit is repeated throughout extended bits
* operands must not be the same size

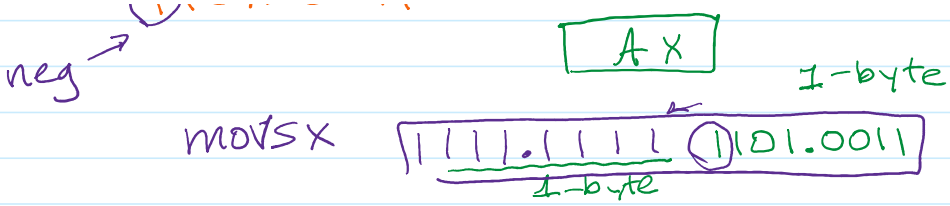
Decimal	Binary
45	0 0 1 0.1 1 0 1 (pos)
	128 64 32 16 . 8 4 2 1

AX: | 0000.0000 | 0010.1101 |

```
  1 1 0 1 0 0 1 0
+               1
-----------------
```

-45 1101.0011 (neg)

AX   1-byte



* Direct Offset Operand

* Add displacement to name of variable

```
.data
    arrayB  byte  10h, 20h, 30h, 40h, 50h
                  +1   +1   +1   +1

.code

    mov  al, arrayB        ; al = 10h
    mov  al, [arrayB+1]    ; al = 20h
    mov  al, [arrayB+3]    ; al = 40h

.data
    arrayW  word  100h, 200h, 300h
                  +2    +2

.code
    mov  ax, [arrayW+2]    ; ax = 200h
    mov  ax, [arrayW+4]    ; ax = 300h

.data
    arrayDW  dword  1000h, 2000h, 3000h

.code
    mov  eax, [arrayDW+4]  ; eax = 2000h
```

INC/DEC Directives

* INC reg/mem
* DEC reg/mem

} increment/decrement by 1

ADD Instruction

* ADD / SUB dest, source
* operands must be same size, and cannot both be memory
* Instruction Pointer Register (IP, EIP, RIP) can't be destination operand

```
ADD   reg, reg
SUB   mem, reg
      reg, mem
      mem, imm
      reg, imm
```

SUB Instruction

* has same restrictions as ADD instruction

NEG Instruction

* reverses sign of value by converting value to its two's complement

```
NEG reg
    mem
```

OFFSET Directive – returns the distance from beginning of enclosing segment

PTR Directive – let's you override operands default size

TYPE Directive - returns size in bytes of an operand for each element in array

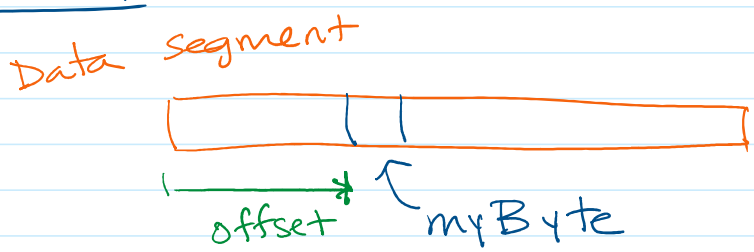
LENGTHOF Directive – returns number of elements in an array

SIZEOF Directive – number of bytes used

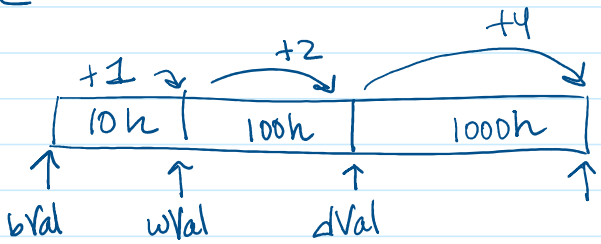
elements in an array

SIZEOF Directive – number of bytes used in an array initializer

OFFSET




```
.data
bVal  byte  10h
wVal  word  100h
dVal  dword 1000h
```




```
.code
mov esi, offset bVal    ; esi = 00404000h  +1
mov esi, offset wVal    ; esi = 00404001h  +2
mov esi, offset dVal    ; esi = 00404003h
```

PTR

```
.data
myDouble dword 12345678h
```

4-byte

low 2 bytes

```
.code
mov ax, myDouble
```

2-byte

```
mov ax, word ptr myDouble
```

moves low word (5678h) due to little-endian order in memory

```
mov ax, word ptr [myDouble+2]
```

moves high word (1234h)

```
mov al, byte ptr myDouble
```

moves low byte (78h)

TYPE

```
.data
var1  byte  ?        ; TYPE var1 = 1
var2  word  ?        ; TYPE var2 = 2
var3  dword ?        ; TYPE var3 = 4

.code
mov eax, TYPE var1
```

LENGTHOF

```
.data
arr1  byte  10,20,30          ; LENGTHOF arr1 = 3
arr2  word  1,2,3,4           ; LENGTHOF arr2 = 4
arr3  byte  "12345678",0      ; LENGTHOF arr3 = 9
```

SIZEOF

product of TYPE & LENGTHOF

sizeOF arr2 = 8

Memory

"1" → 31h
"2" → 32h
⋮
"8" → 38h

Indirect Addressing

* use registers as pointer to access elements of an array
* Indirect operand – operand that uses indirect

* Indirect operand – operand that uses indirect addressing

* Registers that can be used as indirect operands:
  EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

* Surround registers with brackets to dereference address
  * Registers will contain address of some data

Example:

```
.data
    byteVal   byte   10h

.code
    mov  esi , offset byteVal      ; 00404000h
    mov  al , [esi]
```

↳ derefencing a register
* moves into AL the byte pointed to by the address in ESI

Example:

```
.data
    arrayB  byte  10h, 20h, 30h, 40h

.code
    mov esi, offset arrayB
    mov al, [esi]                  ; al=10h
                byte ptr [esi]
    inc esi

    mov al, [esi]                  ; al=20h
            byte ptr [esi]
```

```
mov al, [esi]                    ; al=20h
        byte ptr [esi]
inc esi
mov al, [esi]                    ; al =30h
        byte ptr [esi]
```

* A pointer refers a variable that contains the address of another variable

JMP Instruction

* JMP jumps to a predefined label within your code segment (like a return statement)
* Not a conditional jump

```
.code
    mov eax,0
someLabel:
    inc eax
    jmp someLabel
```
} Infinite loop is engaged

Loop Instruction

* Loop instruction is a conditional jump
* Only jump if ECX is not zero
* ECX is the loop counter
* Don't use Ecx register carelessly when using the loop instruction

Loop

* Decrements ECX by 1
* If ECX ≠ 0, then a jump happens
* If ECX = 0, then you don't jump

* If ECX ≠ 0, then a jump happens
* If ECX = 0, then you don't jump
* precondition for loop is to set ECX to a positive value

```
.code
    mov eax, 0
    mov ecx, 5          ; loop five times

L1:
    add eax, 5
    loop L1             ; decrements ecx by 1
                        ; & jmp if ecx ≠ 0
```

When loop terminates
* EAX will have a value of 25
* ECX will have a value of 0.

sample program

```
.data
    str1 byte "nostra", 0
    lenStr = ($ - str1)
             7   ↑ current location counter
.code
    mov esi, OFFSET str1
    mov ecx, lenStr
    dec ecx             ; ecx=6

L1:
    mov al, [esi]       ; 0001
  * sub al, 10          ; dereference esi
    mov [esi], al       ; AL determines how many bytes to grab from memory
    inc esi
    loop L1
```

	offset	value
str1 →	0000	'n' - 110 → 100 - 'd'
esi →	0001	'o' - 111 → 101 - 'e'
	0002	's' - 115
	0003	't' - 116
	0004	'r' - 114
	0005	'a' - 97 → 87 - 'W'
	0006	0
	$ 0007