Instructions

Data Movements, and Arithmetic Instructions
FLAGS Affected

Constants in program

- Integer Constants
  - Examples: –10, 42d, 10001101b, 0FF3Ah, 777o
  - Radix: b = binary, d = decimal, h = hexadecimal, and o = octal
  - If no radix is given, the integer constant is decimal
  - A hexadecimal beginning with a letter must have a leading 0
- Character and String Constants
  - Enclose character or string in single or double quotes
  - Examples: 'A', "d", 'ABC', "ABC", '4096'
  - Embedded quotes: "single quote ' inside", 'double quote " inside'
  - Each ASCII character occupies a single byte

Lets Learn Instructions

- Data Transfer Instructions
  - Transfer data from/to memory/register
  - Memory to memory transfer in one operation is impossible
  - Examples
    - MOV
    - MOVZX
    - MOVSX
    - XCHG
  - Note:
    - Instructions are written in .CODE section in any procedure

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

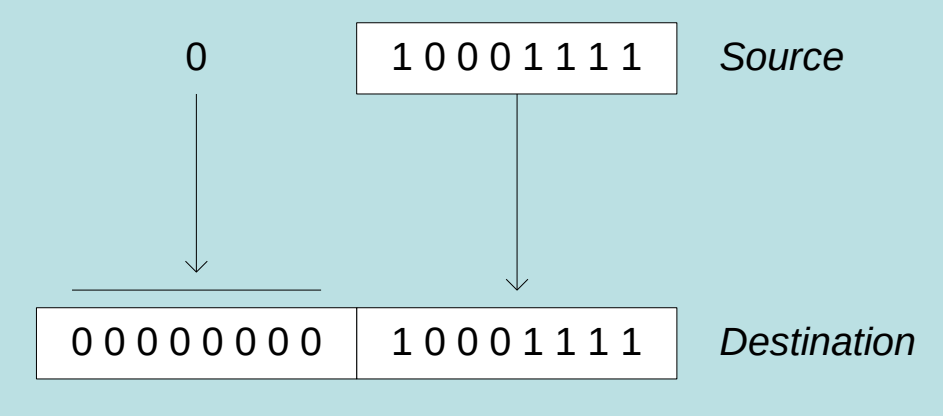
MOV Examples

```
.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
```

Zero Extension

- MOVZX Instruction
  - Fills (extends) the upper part of the destination with zeros
  - Used to copy a small source into a larger destination
  - Destination must be a register

```
movzx r32, r/m8
movzx r32, r/m16
movzx r16, r/m8
```

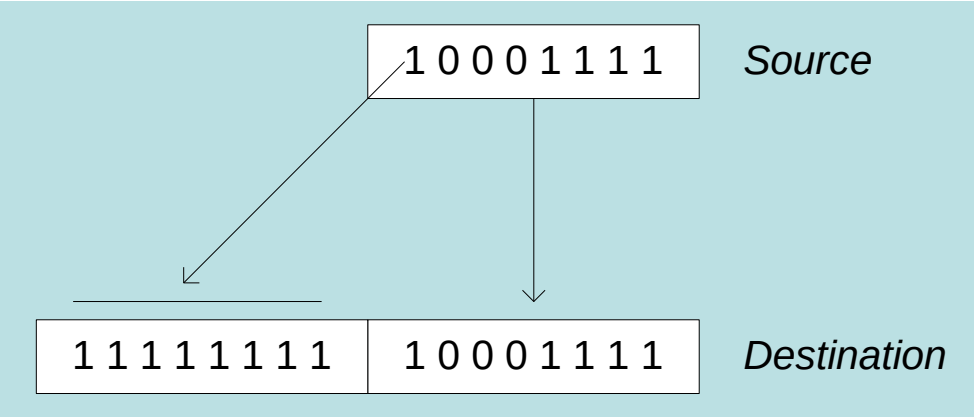



```
mov   bl, 8Fh
movzx ax, bl
```

Sign Extension

- MOVSX Instruction
  - Fills (extends) the upper part of the destination register with a copy of the source operand's sign bit
  - Used to copy a small source into a larger destination

```
movsx r32, r/m8
movsx r32, r/m16
movsx r16, r/m8
```




```
mov   bl, 8Fh
movsx ax, bl
```

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
```

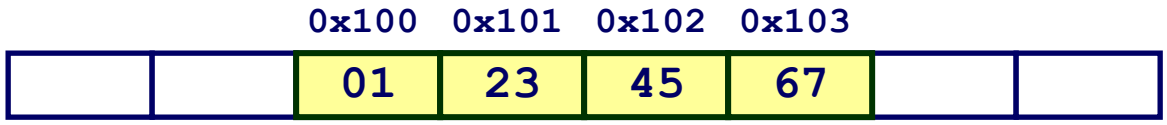
Byte Ordering

- **How should bytes within a multi-byte word be ordered in memory?**
- **Conventions**
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86
    - Least significant byte has lowest address

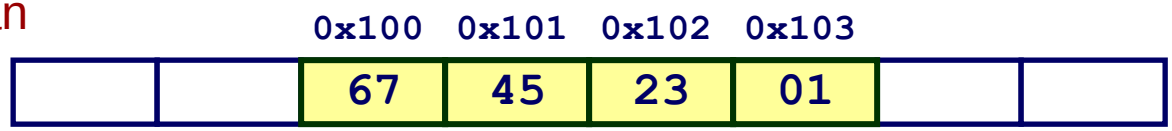
Byte Ordering Example

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

Big Endian



Little Endian



OFFSET Operator

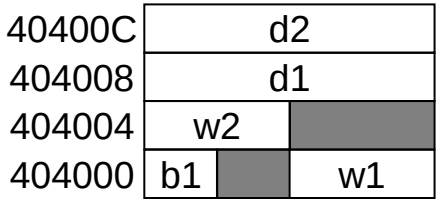
- OFFSET = address of a variable within its segment
  - In FLAT memory, one address space is used for code and data
  - OFFSET = linear address of a variable (32-bit number)

```
.DATA
bVal   BYTE  ?              ; Assume bVal is at 00404000h
wVal   WORD  ?
dVal   DWORD ?
dVal2  DWORD ?

.CODE
mov esi, OFFSET bVal     ; ESI = 00404000h
mov esi, OFFSET wVal     ; ESI = 00404001h
mov esi, OFFSET dVal     ; ESI = 00404003h
mov esi, OFFSET dVal2    ; ESI = 00404007h
```

ALIGN Directive

- ALIGN directive aligns a variable in memory
- Syntax: ALIGN *bound*
  - Where *bound* can be 1, 2, 4, or 16
- Address of a variable should be a multiple of *bound*
- Assembler inserts empty bytes to enforce alignment

```
.DATA          ; Assume that
b1 BYTE   ?    ; Address of b1 = 00404000h
ALIGN 2        ; Skip one byte
w1 WORD   ?    ; Address of w1 = 00404002h
w2 WORD   ?    ; Address of w2 = 00404004h
ALIGN 4        ; Skip two bytes
d1 DWORD  ?    ; Address of d1 = 00404008h
d2 DWORD  ?    ; Address of d2 = 0040400Ch
```



TYPE Operator

- TYPE operator
  - Size, in bytes, of a single element of a data declaration

```
.DATA
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.CODE
mov eax, TYPE var1    ; eax = 1
mov eax, TYPE var2    ; eax = 2
mov eax, TYPE var3    ; eax = 4
mov eax, TYPE var4    ; eax = 8
```

LENGTHOF Operator

- LENGTHOF operator
  - Counts the number of elements in a single data declaration

```
.DATA
array1     WORD     30 DUP(?),0,0
array2     WORD     5 DUP(3 DUP(?))
array3     DWORD    1,2,3,4
digitStr   BYTE    "12345678",0

.code
mov ecx, LENGTHOF array1           ; ecx = 32
mov ecx, LENGTHOF array2           ; ecx = 15
mov ecx, LENGTHOF array3           ; ecx = 4
mov ecx, LENGTHOF digitStr         ; ecx = 9
```

SIZEOF Operator

- SIZEOF operator
  - Counts the number of bytes in a data declaration
  - Equivalent to multiplying LENGTHOF by TYPE

```
.DATA
array1      WORD     30 DUP(?),0,0
array2      WORD     5 DUP(3 DUP(?))
array3      DWORD    1,2,3,4
digitStr    BYTE    "12345678",0

.CODE
mov ecx, SIZEOF array1                  ; ecx = 64
mov ecx, SIZEOF array2                  ; ecx = 30
mov ecx, SIZEOF array3                  ; ecx = 16
mov ecx, SIZEOF digitStr                ; ecx = 9
```

Multiple Line Declarations

A data declaration spans multiple lines if each line (except the last) ends with a comma

The LENGTHOF and SIZEOF operators include all lines belonging to the declaration

```
.DATA
array WORD 10,20,
           30,40,
           50,60

.CODE
mov eax, LENGTHOF array ; 6
mov ebx, SIZEOF array   ; 12
```

In the following example, array identifies the first line WORD declaration only

Compare the values returned by LENGTHOF and SIZEOF here to those on the left

```
.DATA
array  WORD 10,20
       WORD 30,40
       WORD 50,60

.CODE
mov eax, LENGTHOF array ; 2
mov ebx, SIZEOF array   ; 4
```

Arithmetic Instructions

- ADD
- SUB
- INC and DEC
- ADC
- SBB
- MUL
- DIV

✧ Note:

- Instructions are written in .CODE section in any procedure

ADD and SUB Instructions

- ADD *destination*, *source*

  HLL→ *destination* = *destination* + *source*

- SUB *destination*, *source*

  HLL → *destination* = *destination* – *source*

- Destination can be a *register* or a *memory* location
- Source can be a *register*, *memory* location, or a *constant*
- same size
- Memory-to-memory not allowed

Evaluate this . . .

Write a program that adds the following three words:

```
.DATA
array WORD 890Fh,1276h,0AF5Bh
```

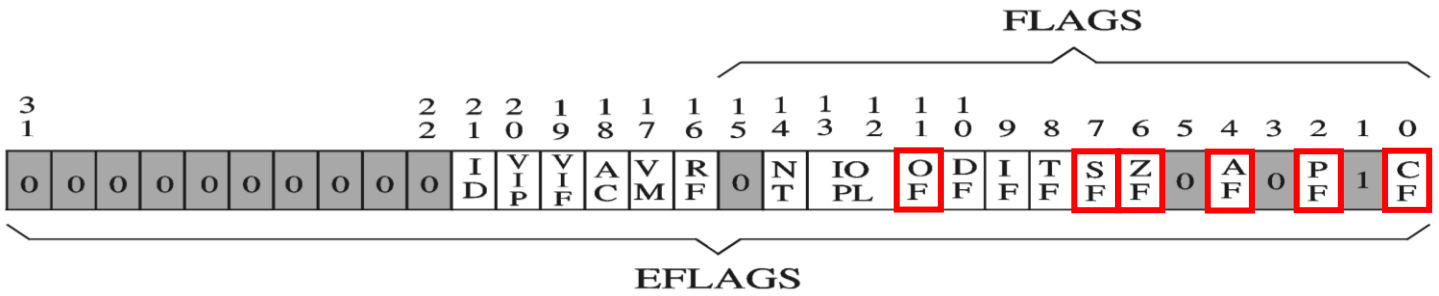
Solution: Accumulate the sum in the AX register

```
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
```

Flags Affected



ADD and SUB affect all the six status flags:

1. Carry Flag: Set when unsigned arithmetic result is out of range
2. Overflow Flag: Set when signed arithmetic result is out of range
3. Sign Flag: Copy of sign bit, set when result is negative
4. Zero Flag: Set when result is zero
5. Auxiliary Carry Flag: Set when there is a carry from bit 3 to bit 4
6. Parity Flag: Set when parity in least-significant byte is even

Hardware Viewpoint

- CPU cannot distinguish signed from unsigned integers
  - YOU, the programmer, give a meaning to binary numbers
- How the ADD instruction modifies OF and CF:
  - CF = (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)
- Hardware does SUB by …
  - ADDing destination to the 2's complement of the source operand
- How the SUB instruction modifies OF and CF:
  - Negate (2's complement) the source and ADD it to destination
  - OF = (carry out of the MSB) XOR (carry into the MSB)
  - CF = INVERT (carry out of the MSB)

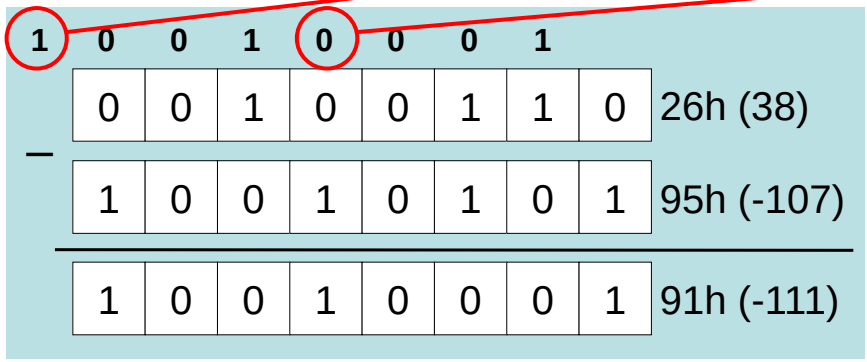
MSB = Most Significant Bit

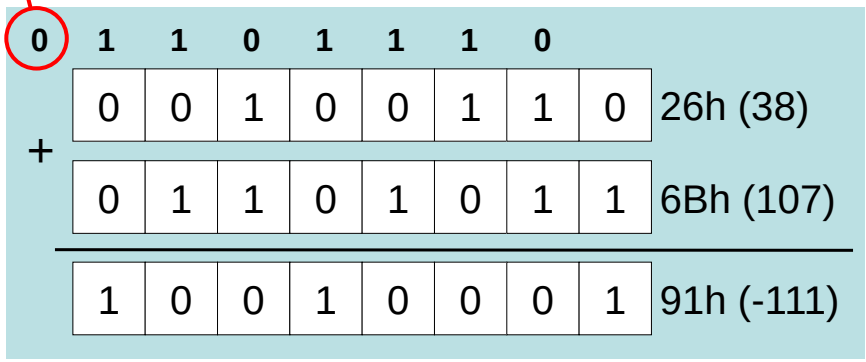
XOR = eXclusive-OR operation

ADD and SUB Examples

For each of the following marked entries, show the values of the destination operand and the six status flags:

```
mov al,0FFh     ; AL=-1
add al,1        ; AL=00h    CF=1 OF=0 SF=0 ZF=1 AF=1 PF=1
sub al,1        ; AL=FFh    CF=1 OF=0 SF=1 ZF=0 AF=1 PF=1
mov al,+127     ; AL=7Fh
add al,1        ; AL=80h    CF=0 OF=1 SF=1 ZF=0 AF=1 PF=0
mov al,26h
sub al,95h      ; AL=91h    CF=1 OF=1 SF=1 ZF=0 AF=0 PF=0
```





INC, DEC, and NEG Instructions

- INC *destination*
  - *destination* = *destination* + 1
  - More compact (uses less space) than: ADD *destination*, 1
- DEC *destination*
  - *destination* = *destination* – 1
  - More compact (uses less space) than: SUB *destination*, 1
- NEG *destination*
  - *destination* = 2's complement of *destination*
- Destination can be 8-, 16-, or 32-bit operand
  - In memory or a register
  - NO immediate operand

Affected Flags

- INC and DEC affect five status flags
  - Overflow, Sign, Zero, Auxiliary Carry, and Parity
  - Carry flag is NOT modified
- NEG affects all the six status flags
  - Any nonzero operand causes the carry flag to be set

```
.DATA
   B SBYTE -1          ; 0FFh
   C SBYTE 127         ; 7Fh
.CODE
   inc B   ; B=0              OF=0 SF=0 ZF=1 AF=1 PF=1
   dec B   ; B=-1=FFh         OF=0 SF=1 ZF=0 AF=1 PF=1
   inc C   ; C=-128=80h       OF=1 SF=1 ZF=0 AF=1 PF=0
   neg C   ; C=-128   CF=1    OF=1 SF=1 ZF=0 AF=0 PF=0
```

ADC and SBB Instruction

- ADC Instruction: Addition with Carry

  ADC *destination, source*

  ***destination* = *destination* + *source* + CF**

- SBB Instruction: Subtract with Borrow

  SBB *destination, source*

  ***destination* = *destination* - *source* – CF**

- 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

Extended Arithmetic

- ADC and SBB are useful for extended arithmetic
- Example: 64-bit addition
  - Assume first 64-bit integer operand is stored in EBX:EAX
  - Second 64-bit integer operand is stored in EDX:ECX
- Solution:

```
add eax, ecx  ;add lower 32 bits
adc ebx, edx  ;add upper 32 bits + carry
```

  64-bit result is in EBX:EAX

- STC and CLC Instructions
  - Used to Set and Clear the Carry Flag

Thanks!