بسم اللہ الرحمن الرحیم

EE213 COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE

Fall 2017

4

Data Transfers, Addressing, and Arithmetic

OUTLINES

- Data Transfer Instructions
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions

4.1 DATA TRANSFER INSTRUCTIONS

Operand Types

- Instructions in assembly language can have zero, one, two, or three operands.

- `mnemonic`

- `mnemonic [destination]`

- `mnemonic [destination],[source]`

- `mnemonic [destination],[source1],[source2]`

- The three types of operands are:

1. **Immediate :** a numeric literal expression /a constant integer (8, 16, or 32 bits), value is encoded within the instruction

2. **Register:** the name of a register, register name is converted to a number and encoded within the instruction

3. **Memory:** references a location in memory, memory address is encoded within the instruction, or a register holds the address of a memory location

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	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
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

```
mov al var1

A0 00010400
```

Direct Memory Operands

- Variable names are references to offsets within the data segment.
- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

```
.data
var1 BYTE 10h
.code
mov al,var1                   ; AL = 10h
mov al,[var1]                 ; AL = 10h
```

alternate format

MOV INSTRUCTION

- The MOV instruction copies data from a source operand to a destination operand. Known as a *data transfer* instruction.

```
MOV destination,source
```

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer register (IP, EIP) and CS cannot be a destination operand.

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
```

```
.data
     count BYTE 100
      wVal  WORD 2

.code
     mov bl,count
     mov ax,wVal
     mov count,al

     mov al,wVal          ; error
     mov ax,count         ; error
     mov eax,count        ; error
```

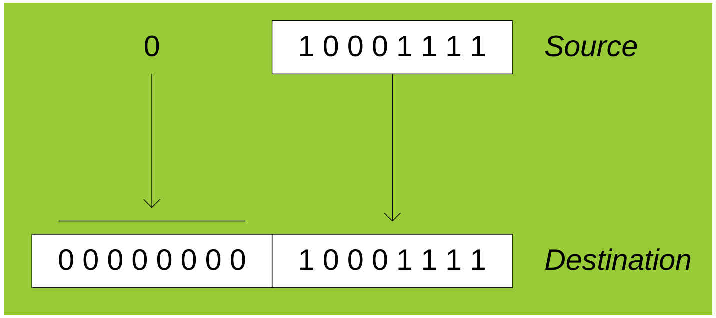
Zero Extension

- MOV instruction cannot directly copy data from a smaller operand to a larger one.

```
mov bl,10001111b
mov ax,bl                                 ; error
```

- MOVZX (move with zero-extend) instruction fills (extends) the upper half of the destination with zeros.

```
mov bl,10001111b
movzx ax,bl      ; zero-extension
```

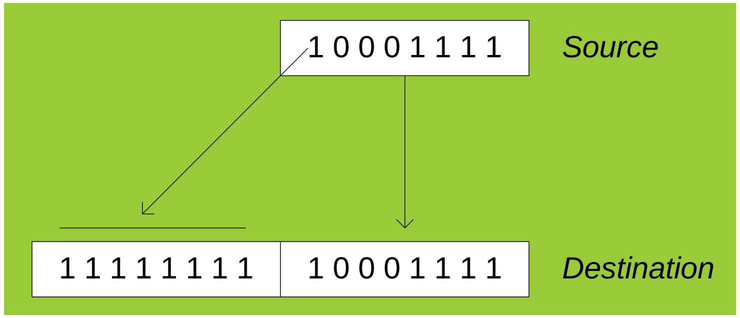


```
.data
        byte1 BYTE 9Bh
        word1 WORD 0A69Bh
.code
        movzx eax,word1             ; EAX = 0000A69Bh
        movzx edx,byte1             ; EDX = 0000009Bh
        movzx cx,byte1              ; CX = 009Bh
```

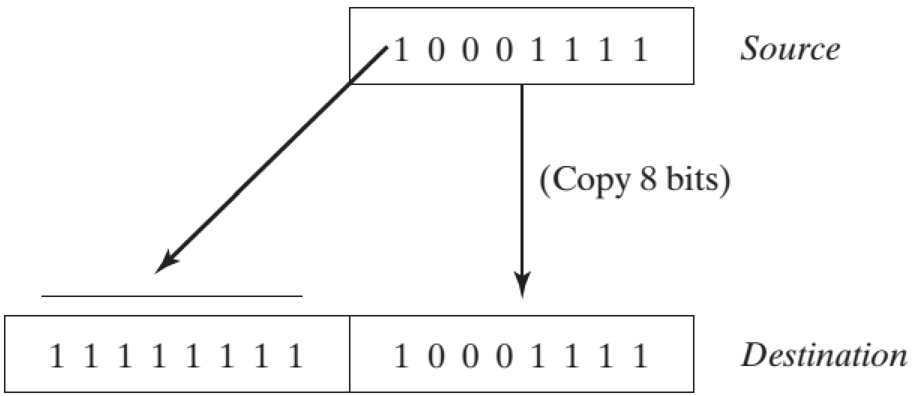
MOVSX Instruction

- The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and fills the upper half of the destination with a copy of the source operand's sign bit.

```
mov bl,10001111b
movsx ax,bl     ; sign extension
```



```
.data
      byteVal BYTE 10001111b
.code
      movsx ax,byteVal      ; AX = 1111111110001111b
```



XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

```
.data
    var1 WORD 1000h
    var2 WORD 2000h

.code
    xchg ax,bx              ; exchange 16-bit regs
    xchg ah,al              ; exchange 8-bit regs
    xchg var1,bx            ; exchange mem, reg
    xchg eax,ebx            ; exchange 32-bit regs

    xchg var1,var2          ; error: two memory operands
```

Direct-Offset Operands lets you access memory locations that may not have explicit labels.

- A constant is added to a data label to produce an *effective address* (EA). The address is dereferenced to get the value inside its memory location.

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

.code
   mov al,arrayB+1               ; AL = 20h
   mov al,[arrayB+1]             ; alternative notation
```

4.2 ADDITION AND SUBTRACTION

INC and DEC Instructions

- The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a register or memory operand.

```
.data
    myWord  WORD 1000h
    myDword DWORD 10000000h
.code
    inc myWord                  ; 1001h
    dec myWord                  ; 1000h
    inc myDword                 ; 10000001h

    mov ax,00FFh
    inc ax                      ; AX = 0100h
    mov ax,00FFh
    inc al                      ; AX = 0000h
```

ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size.

ADD *dest,source*

SUB Instruction

The SUB instruction subtracts a source operand from a destination operand

SUB *dest,source*

- The set of possible operands is the same as for the MOV instruction

```
.data
    var1 DWORD 10000h
    var2 DWORD 20000h

.code                           ; ---EAX---
    mov eax,var1                ; 00010000h
    add eax,var2                ; 00030000h
    add ax,0FFFFh               ; 0003FFFFh
    add eax,1                   ; 00040000h
    sub ax,1                    ; 0004FFFFh
```

NEG Instruction

The NEG (negate) instruction reverses the sign of a number by converting the number to its two's complement

```
NEG reg
NEG mem
```

Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

```
Rval = -Xval + (Yval – Zval)
```

```
    Rval DWORD ?
    Xval DWORD 26
    Yval DWORD 30
    Zval DWORD 40

.code
    mov eax,Xval
    neg eax                 ; EAX = -26

    mov ebx,Yval
    sub ebx,Zval            ; EBX = -10

    add eax,ebx
    mov Rval,eax            ; -36
```

Flags Affected by Addition and Subtraction

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand

- We use the values of CPU status flags to check the outcome of arithmetic operations and to activate conditional branching instructions.

- Essential flags:
  - **Zero flag –** set when destination equals zero
  - **Sign flag –** set when destination is negative; if the MSB of the destination operand is set,
  - **Carry flag –** set when unsigned value is out of range
  - **Overflow flag –** set when signed value is out of range

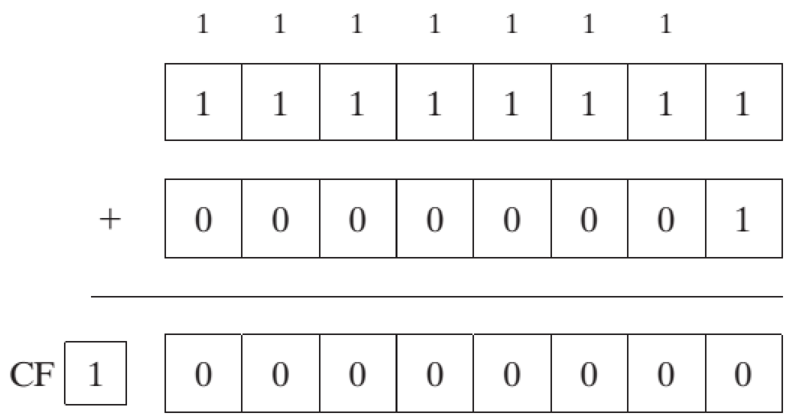
```
mov cx,1
sub cx,1                ; CX = 0, ZF = 1

mov ax,0FFFFh
inc ax                  ; AX = 0, ZF = 1
inc ax                  ; AX = 1, ZF = 0

mov cx,0
sub cx,1                ; CX = -1, SF = 1
add cx,2                ; CX = 1, SF = 0
```

Addition and the Carry Flag

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



Subtraction and the Carry Flag

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

		0	0	0	0	0	0	0	1	(1)
	+	1	1	1	1	1	1	1	0	(−2)
CF	1	1	1	1	1	1	1	1	1	(FFh)

Signed Operations: Sign and Overflow Flags

- The **Sign flag** is set when the result of a signed arithmetic operation is negative.

```
mov eax,4
sub eax,5          ; EAX = -1, SF = 1
```

- The **Overflow flag** is set when the result of a signed arithmetic operation overflows or underflows the destination operand.

```
mov al,127
add al,1           ; OF = 1
```

```
mov al,-128
sub al,1           ; OF = 1
```

- The NEG instruction produces an invalid result if the destination operand cannot be stored correctly.

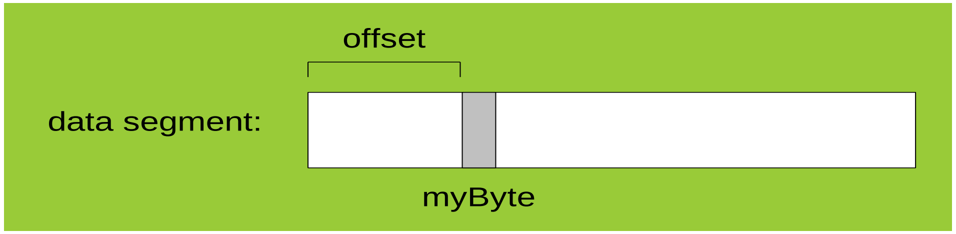
```
mov al,-128 ; AL = 10000000b
neg al      ; AL = 10000000b, OF = 1
```

The Overflow flag is set, indicating that AL contains an invalid value:

4.3 DATA-RELATED OPERATORS AND DIRECTIVES

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

- The **OFFSET** operator returns the offset of a data label.
  - returns the distance (in bytes) of a variable from the beginning of its enclosing segment.



Assuming, data segment begins at 00404000h:

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

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

- You can use the **PTR** operator to override the declared size of an operand.

```
.data
        myDouble DWORD 12345678h
.code
        mov ax,myDouble                 ;error - why?
        mov ax,WORD PTR myDouble        ; loads 5678h
```

- Why wasn't **1234h** moved into **AX?** x86 processors use the *little endian* storage format in which the low-order byte is stored at the variable's starting address.

- The **TYPE** operator returns the size, in bytes, of a single element of a data declaration.

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

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

- The **LENGTHOF** operator counts the number of elements in an array, defined by the values appearing on the same line as its label.

```
.data
        byte1 BYTE 10,20,30
        array1 WORD 30 DUP(?),0,0
        array2 WORD 5 DUP(3 DUP(?))
        array3 DWORD 1,2,3,4
        digitStr BYTE "12345678",0
```

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

- If you declare an array that spans multiple program lines, **LENGTHOF** only regards the data from the first line as part of the array (here **LENGTHOF myArray** returns 5).

```
myArray BYTE 10,20,30,40,50
        BYTE 60,70,80,90,100
```

```
.data                                             LENGTHOF
    byte1  BYTE 10,20,30                          ; 3
    array1 WORD 30 DUP(?),0,0                     ; 32
    array2 WORD 5 DUP(3 DUP(?))                   ; 15
    array3 DWORD 1,2,3,4                          ; 4
    digitStr BYTE "12345678",0                    ; 9

.code
    mov ecx, LENGTHOF array1                      ; 32
```

- The **`SIZEOF`** operator returns a value that is equivalent to multiplying **`LENGTHOF`** by **`TYPE`**.

```
                                                  SIZEOF
.data
    byte1  BYTE 10,20,30                          ; 3
    array1 WORD 30 DUP(?),0,0                     ; 64
    array2 WORD 5 DUP(3 DUP(?))                   ; 30
    array3 DWORD 1,2,3,4                          ; 16
    digitStr BYTE "12345678",0                    ; 9

.code
    mov ecx, SIZEOF array1                        ; 64
```

The **LABEL** directive assigns an alternate label name and type to an existing storage location. **LABEL** does not allocate any storage of its own.

- A common use of **LABEL** is to provide an alternative name and size attribute for the variable declared next in the data segment.

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

```
.data
    LongValue LABEL DWORD
    val1 WORD 5678h
    val2 WORD 1234h
.code
    mov eax,LongValue ; EAX = 12345678h
```

4.4 INDIRECT ADDRESSING

- Direct addressing is rarely used for array processing because it is impractical to use constant offsets to address more than a few array elements.
  - An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).

```
.data
    val1 BYTE 10h,20h,30h
.code
    mov esi,OFFSET val1
    mov al,[esi]                ; dereference ESI (AL = 10h)

    inc esi
    mov al,[esi]                ; AL = 20h

    inc esi
    mov al,[esi]                ; AL = 30h
```

- The size of an operand may not be evident from the context of an instruction.

```
inc [esi]   ; error: operand must have size
```

  - Because the assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

```
inc BYTE PTR [esi]
```

Arrays

- Indirect operands are ideal tools for stepping through 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
```

Indexed Operands

- An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

[*label* + *reg*] **_label_[*reg*]**

```
.data
    arrayW WORD 1000h,2000h,3000h

.code
    mov esi,0
    mov ax,[arrayW + esi]          ; AX = 1000h

    mov ax, arrayW[esi]            ; alternate format

    add esi,2
    add ax,[arrayW + esi]
```

4.5 JMP AND LOOP INSTRUCTIONS

- By default, the CPU loads and executes programs sequentially, however, control may be transferred to a new location in the program.

- A *transfer of control*, or *branch*, is a way of altering the order in which statements are executed, there are two basic types:

1. **Unconditional Transfer**: No condition is involved, control is transferred to a new location in all cases.

2. **Conditional Transfer:** The program branches if a certain condition is true (based on status of flags).

JMP Instruction

The `JMP` instruction causes an unconditional transfer to a destination, identified by a code label.

```
JMP destination
```

offset of *`destination`* is moved into the instruction pointer, causing execution to continue at the new location

```
top:    INC AX
        MOV BX, AX
        jmp top
```

LOOP Instruction

The `LOOP` instruction, formally known as *Loop According to ECX Counter*, repeats a block of statements a specific number of times.

- ECX is automatically used as a counter and is decremented each time the loop repeats.

```
LOOP destination
```

The loop destination must be within -128 to +127 bytes of the current location counter.

- –128 bytes is the largest backward jump from current instruction +127 bytes is the largest forward jump.

The execution of the LOOP instruction involves two steps:

1. First, it subtracts 1 from ECX.
2. Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by *destination*. Otherwise, no jump takes place, and control passes to the instruction following the loop.

```
mov ax,0
mov ecx,5

L1:    inc ax
       loop L1
       mov bx,ax
```

YOUR TURN . . .

What will be the value of BX?

```
   mov ax,6
   mov ecx,4
L1:
   inc ax
   loop L1
   mov bx, ax
```

SUMMARY

Data Transfer

- MOV – data transfer from source to destination
- MOVSX, MOVZX, XCHG

Operand types

- direct, direct-offset, indirect, indexed

Arithmetic

- INC, DEC, ADD, SUB, NEG
- Sign, Carry, Zero, Overflow flags

Operators

- OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, LABEL

JMP and LOOP – branching instructions