Assembly Language

Chapter 3 & 4

Slides prepared by Kip Irvine and modified by Dr. Karim Emara and Dr. Salsabil Amin

Agenda

- CF and OF Flags ADD/SUB
- INC and DEC Instructions
- Data Operators
- Indirect Addressing

Flags

- Why flags are important:
  - To detect calculation errors
  - Check the outcome of an arithmetic operation
  - Activate conditional branching (loops and if/else)

- Let’s see how each flag is affected by ADD/SUB instructions.

Carry Flag (CF)

- CF is set when the result of an **unsigned** operation is out of destination range
- ADD: CF is set when result is larger than the dest operand

```
mov al, FFh  ; 255
add al, 1    ; CF = 1, SUM too large for AL
```

- SUB: CF is set when subtract a large operand from a smaller one (i.e., there's a borrow bit)

```
mov al, 3
sub al, 5 ; CF = 1, Res(-2)
 cannot be placed in
 unsigned number
```

00000011	00000011
- 00000101	+11111011
------------	------------
	11111110
	Carry out=0

Overflow Flag (OF)

- OF is set when the result of a **signed** operation is out of destination range

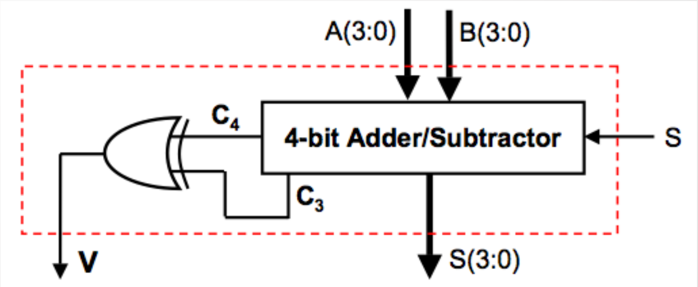
```
mov al, +127
add  al, 1  ; OF = 1,    AL = 80h (-128)
```

- The overflow flag is set when . . .
  - Two positive operands are added and their sum is negative → Ex: +127 +1
  - Two negative operands are added and their sum is positive → EX: -128 -1
  - Overflow cannot occur when adding operands of opposite signs

Hardware Perspective

- All CPU instructions operate exactly the same on signed and unsigned integers
- CPU does NOT distinguish signed from unsigned
- Hardware-wise, the CF and OF flags are set according to the following:
  - CF = Carry out for **ADD** instruction
    = INVERT(Carry out) for **SUB** instruction *
  - OF = (carry out of the MSB) XOR (carry in the MSB)

*[AF is inverted with SUB as well]



p	q	$p \oplus q$
1	1	0
1	0	1
0	1	1
0	0	0

(board)

Add

255 +
1

OF=0

CF=1

1 1 1 1 1 1 1 1

1111 1111
0000 0001 +
0000 0000

Sub

128
− 1

128
+(−1)

OF=1

CF=0

1 0

1000 0000
1111 1111 +
0111 1111

	U	S	
CF =1	255 + 1 = 256	−1 + 1 = 0	OF =0

	U	S	
CF =0	128 − 1 = 127	−128 − 1 = −129	OF =1

Overflow Flag (OF)

- What will be the values of the Overflow flag?
  - `mov al, 80h`
  - `add al, 92h` ;al= 12h OF = 1

-ve operands, +ve result

	1	0	0	0	0	0	0	0	
	1	0	0	0	0	0	0	0	80h (-128)
+	1	0	0	1	0	0	1	0	92h (-110)
CF=1	0	0	0	1	0	0	1	0	12h result should be -238 < -128

Overflow Flag (OF)

- What will be the values of the Overflow flag?
  - `mov al, 40h`
  - `add al, 45h` ;al= 85h OF = 1

+ve operands, -ve result

	0	1	0	0	0	0	0	0		
		0	1	0	0	0	0	0	0	40h (64)
	+	0	1	0	0	0	1	0	1	45h (69)
CF=0		1	0	0	0	0	1	0	1	85h result should be 133 > 127

Overflow Flag (OF)

- What will be the values of the Overflow flag?
  - **mov al, -128**
  - **sub al, 1**

-ve operands, +ve result

;al= +127 OF = 1

	1	0	0	0	0	0	0	0	
		1	0	0	0	0	0	0	0	80h (-128)
+		1	1	1	1	1	1	1	1	FFh (-1)
CF=0		0	1	1	1	1	1	1	1	result should be -129 < -128

Addition and Subtraction

- Show the values of the destination operand and the six status flags
- Notice : AF is inverted in SUB like CF

```
mov al,0FFh     ; AL=-1  MOV does NOT affect flags
add al,1        ; AL=       CF=  OF=  SF=  ZF=  AF=  PF=
sub al,1        ; AL=       CF=  OF=  SF=  ZF=  AF=  PF=
mov al,+127     ; AL=7Fh
add al,1        ; AL=       CF=  OF=  SF=  ZF=  AF=  PF=
mov al,26h
sub al,95h      ; AL=       CF=  OF=  SF=  ZF=  AF=  PF=
```

result=0
⇒ zf=1

12

```
mov al,0FFh      ; AL=FFh   CF=X OF=X SF=X ZF=X AF=X PF=X
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   CF=X OF=X SF=X ZF=X AF=X PF=X
add al,1         ; AL=80h   CF=0 OF=1 SF=1 ZF=0 AF=1 PF=0
mov al,26h                  X    X    X    X    X    X
sub al,95h       ; AL=91h   CF=1 OF=1 SF=1 ZF=0 AF=0 PF=0
```

Add al,1
(FF + 1)

	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0

Add al,1
(127 + 1)

	0	1	1	1	1	1	1	1
	0	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0

Sub al,1
(0 - 1)

$0 - 1$ $\quad$ $0 + (-1)$

	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
+	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1

Sub al,95h
(26h – 95h)

	0	0	1	0	0	1	1	0
-	1	0	0	1	0	1	0	1

OF=1

CF=1

	0	1	1	1	1	1		
	0	0	1	0	0	1	1	0
+	0	1	1	0	1	0	1	1
	1	0	0	1	0	0	0	1

91h

Addition and Subtraction

- Show the values of the destination operand and the six status flags
- Notice : AF is inverted in SUB like CF

```
mov al,0FFh     ; AL=-1   MOV does NOT affect flags
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=-128    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
```

1	0	0	1	0	0	0	1		
	0	0	1	0	0	1	1	0	26h (38)
–	1	0	0	1	0	1	0	1	95h (-107)

0	1	1	0	1	1	1	0		
	0	0	1	0	0	1	1	0	26h (38)
+	0	1	1	0	1	0	1	1	6Bh (107)
	1	0	0	1	0	0	0	1	91h (-111)

Agenda

- CF and OF Flags ADD/SUB
- INC and DEC Instructions
- Data Operators
- Indirect Addressing

Addition and Subtraction

- INC, DEC, NEG instructions
  - INC ***dest***
    - Like **dest ++** in C language
    - More compact (uses less space) than: ADD dest, 1
  - DEC ***dest***
    - Like **dest --** in C language
    - More compact (uses less space) than: SUB dest, 1
  - NEG ***dest***
    - **dest** = 2's complement of destination (0-dest)

Addition and Subtraction

- ***dest*** can be
  - 8-, 16-, or 32-bit operand
  - Memory or a register
  - NO immediate operand

Addition and Subtraction

- INC and DEC affect <u>five</u> status flags
  - Overflow, Sign, Zero, Auxiliary Carry, and Parity similar to ADD and SUB instruction
  - Carry flag is **NOT** modified
- NEG affects all the six status flags
  - Any **nonzero source operand** causes the **carry flag to be set**
  - **OF** is set if the result cannot be represented in the operand
    - Maximum negative number in range (e.g. +128 in 8-bit register)

(board)

$\text{neg } x \equiv 0 - x$

if x $\xrightarrow{0}$ $0-0 \Rightarrow$ no borrow $\Rightarrow \boxed{CF=0}$

if x $\xrightarrow{\text{non zero}}$ borrow $\Rightarrow \boxed{CF=1}$

---

neg $\boxed{-128}$, in 8-bits range $-128 \to 127$

128 : 1000 0000 > 127

-128 : 1000 0000

$\Downarrow$

can not be represented in 8-bits

INC/DEC/NEG

- Trace the flags

```
.DATA
    B SBYTE -1          ; 0FFh  or  1111 1111
    C SBYTE 127         ; 7Fh   or  0111 1111
.CODE
    inc B  ; B=0       CF=X OF=0 SF=0 ZF=1 PF=1
    dec B  ; B=-1      CF=X OF=0 SF=1 ZF=0 PF=1
    inc C  ; C=-128=80h     OF=1 SF=1 ZF=0 PF=0
    neg C  ; C=-128    CF=1 OF=1 SF=1 ZF=0 PF=0
```

non-zero   max -ve number

0	1	1	1	1	1	1	1		
	0	1	1	1	1	1	1	1	127
+	0	0	0	0	0	0	0	1	1+
	1	0	0	0	0	0	0	0	128-

← inc C

← Considered as -128

Agenda

- CF and OF Flags ADD/SUB
- INC and DEC Instructions
- Data Operators
- Indirect Addressing

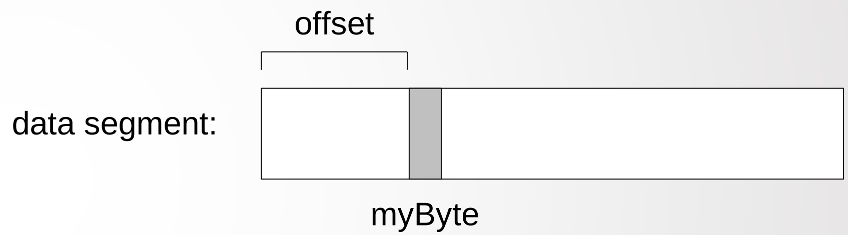
Data-Related Operators and Directives

1. OFFSET Operator
2. PTR Operator
3. TYPE Operator
4. LENGTHOF Operator
5. SIZEOF Operator
6. LABEL Directive

OFFSET operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
- Let's assume that the 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
```



bval		00404000
wval		00404001
		00404002
dval		00404003
		00404004
		00404005
		00404006
dval2		00404007

OFFSET operator

- The value returned by OFFSET is a pointer

// C++ version:

```
char array[1000];
char * p = array;
```

; Assembly language:

```
.data
 array BYTE 1000 DUP(?)
.code
 mov esi, OFFSET array
```

PTR operator

- PTR operator overrides the default type of a label (variable). Provides the flexibility to access part of a variable
- Similar to typecasting in C

```
.data
myDouble DWORD 12345678h
.code
mov ax, myDouble                ; error – why?
mov ax, WORD PTR myDouble       ; AX = 5678h
mov WORD PTR myDouble, 3412h ; myDouble=12343412h
```

myDouble		
	78	0x4020
	56	0x4021
	34	0x4022
	12	0x4023

PTR operator

```
.data
   myDouble DWORD 12345678h
.code
   mov al,BYTE PTR  myDouble       ;AL=78h
   mov al,BYTE PTR [myDouble+1]    ;AL=56h
   mov al,BYTE PTR [myDouble+2]    ;AL=34h
   mov ax,WORD PTR  myDouble       ;AX=5678h
   mov ax,WORD PTR [myDouble+2]    ;AX=1234h
```

myDouble	78	0x4020
	56	0x4021
	34	0x4022
	12	0x4023

PTR operator

- PTR can be used to **combine elements** of a smaller data type and move them into a larger operand
- The CPU will automatically reverse the bytes

```
.data
   myBytes BYTE 12h,34h,56h,78h
.code
   mov ax, WORD PTR [myBytes]     ;AX=3412h
   mov ax, WORD PTR [myBytes+2]   ;AX=7856h
   mov eax,DWORD PTR myByte       ;EAX=78563412h
```

myBytes	12
	34
	56
	78

PTR operator – Your turn

- **Ex: Write down the value of each destination**

```
.data
varB BYTE 65h, 31h, 02h, 05h
varW WORD 6543h, 1202h
varD DWORD 12345678h
.code
mov ax, WORD PTR [varB+2]        ;0502h
mov bl, BYTE PTR varD            ;78h
mov bl, BYTE PTR [varW+2]        ;02h
mov ax, WORD PTR [varD+2]        ;1234h
mov eax,DWORD PTR varW           ;12026543h
```

varB	65
	31
	02
	05
varW	43
	65
	02
	12
varD	78
	56
	34
	12

TYPE operator

- 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 ?
.code
mov eax, TYPE var1          ; 1
mov eax, TYPE var2          ; 2
mov eax, TYPE var3          ; 4
mov eax, TYPE var4          ; 8
```

Similar to ??

LENGTHOF operator

- The LENGTHOF operator counts the **number of elements** in a single data declaration

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

SIZEOF operator

- The SIZEOF operator returns number of **bytes** of a data declaration (i.e., LENGTHOF * TYPE)

```
.data
array1 WORD 30 DUP(?), 0, 0        ; 64 →32*2
array2 WORD 5 DUP(3 DUP(10))       ; 30 →15*2
array3 DWORD 10, 20,
             30, 40,               ;one array
             50, 60                ;24 → 6*4
.code
mov ax, SIZEOF array3             ; 24
mov bx, LENGTHOF array3           ; 6
mov cx, TYPE array3               ; 4
```

SIZEOF operator

- Caution: Array spanning multiple lines

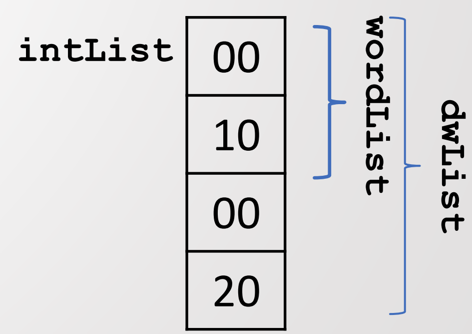
```
.data
array   WORD 10, 20
        WORD 30
        WORD 50, 60, 70
.code
mov eax, LENGTHOF array   ; 2
mov ebx, SIZEOF array     ; 4
```

This is NOT a single array
These are 3 different array definitions

LABEL directive

- Assigns an alternate label and type to the next storage location
- LABEL does **NOT** allocate any storage of its own

```
.data
dwList    LABEL DWORD  ;no declaration
wordList LABEL WORD;no declaration
intList  BYTE 00h, 10h, 00h, 20h
.code
mov eax, dwList  ; 20001000h
mov cx, wordList; 1000h
mov dl, intList  ; 00h
```



LABEL directive

```
.data
dwList   LABEL DWORD
arr1     byte 1, 2, 3, 4
byteList LABEL byte
var1     DWORD 01020304h
.code
mov eax, dwList          ; 04030201h
mov al, byteList         ; 04h
mov dl, byteList+3       ; 01h
```

dwList→	01	arr1
	02	
	03	
	04	
byteList→	04	var1
	03	
	02	
	01	

Agenda

- CF and OF Flags ADD/SUB
- INC and DEC Instructions
- NEG Instruction
- Data Operators
- Indirect Addressing

Remember Direct Offset Addressing?

- **Direct Offset Addressing** by adding a constant offset (displacement) to the label to produce the effective address

```
.data
  arrayB BYTE 10h,20h,30h,40h
.code
  mov al, arrayB + 1  ; AL = 20h
  mov al,[arrayB +1]  ;alternative notation
  mov al, arrayB[1]  ;yet another notation
```

- **Note:** offset expresses on **bytes** not an item index

Register-indirect Addressing

- Direct offset mode is not so flexible since it is not practical to traverse an array using **constant offsets**

- Register-indirect addressing mode solves this problem by allowing storing the array address in a **register** (similar to pointers in C)

Register-indirect Addressing

- The operand offset is stored in a register
- Brackets [ ] are used to surround the register holding the address
- For 32-bit addressing, any 32-bit register can be used

```
mov ebx, OFFSET array1 ;ebx = array1 address
mov eax, [ebx]         ;[ebx] gets 1st item
```

Register-indirect Addressing

- In this mode, register holds the address of a variable, usually an **array** or **string**.

```
.data
    var1 BYTE 10h, 20h, 30h
.code
mov esi, OFFSET var1
mov al, [esi]          ; AL = 10h
inc esi                ; esi ++
mov al, [esi]          ; AL = 20h
inc esi                ; esi ++
mov al, [esi]          ; AL = 30h
```

*

Register-indirect Addressing

- Register-indirect Addressing can point to any size of a memory location (byte, word, ..etc)
- Thus, you should use PTR to clarify the size of a memory operand

```
.data
myCount WORD 0
.code
mov esi, OFFSET myCount
inc [esi]    ;Error: operand must have size
inc WORD PTR [esi]    ; OK
```

inc esi
inc [esi]
⇒ diff ??

Register-indirect Addressing

- This addressing mode is ideal for traversing an array.
- Note that the register in brackets must be incremented by a value that matches the **array type**
  - 1 for BYTE,
  - 2 for WORD,
  - 4 for DWORD

Register-indirect Addressing

- Write a program to **sum** the array elements

```
.data
    arrayW WORD 1000h, 2000h, 3000h
.code
mov esi, OFFSET arrayW
mov ax, [esi]
add esi, 2      ; or: add esi, TYPE arrayW
add ax, [esi]
add esi, 2
add ax, [esi]  ; AX = sum of the array
```

Or we can write: `add esi, type arrayW`

Register-indexed Addressing

- Register-indexed operand treats the register value as an index added to the array offset to generate the effective address.
- There are two notational forms:

[label + reg] or **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]
```

Register-indexed Addressing

- You can **scale** the indexed operand to the offset of an array element. This is done by multiplying the index by the array's TYPE

```
.data
arrB BYTE  0, 1, 2, 3, 4, 5
arrW WORD  0, 1, 2, 3, 4, 5
arrD DWORD 0, 1, 2, 3, 4, 5
.code
mov esi, 4          ;item index (0-based)
mov al, arrB[esi*TYPE arrB] ; arrW[esi*1]    ;arrW[4]
mov ax, arrW[esi*TYPE arrW] ; arrW[esi*2]    ;arrW[8]
mov eax, arrD[esi*TYPE arrD]; arrD[esi*4]    ;arrW[16]
```

- All instructions access item of index 4

Reference

- Textbook:
  - Chapter 4: 4.1 – 4.4

Thank you ☺