x86 Assembly Programming Part 3

EECE416 Microcomputer

Charles Kim
Howard University

Resources:

Intel 80386 Programmers Reference Manual
Essentials of 80x86 Assembly Language
Introduction to 80x86 Assembly Language Programming

Exercise of Register Size and Data –
Do it by hand or by Coding

- Example:

Before *Instruction* *After*

EAX: 01 1F F1 23 → `mov AX, -1` → EAX: 01 1F FF FF

	Before	*Instruction*	*After*
(a)	EBX: 00 00 FF 75		
	ECX: 00 00 01 A2	`mov ebx, ecx`	EBX, ECX
(b)	EAX: 00 00 01 A2	`mov eax, 100`	EAX
(c)	EDX: FF 75 4C 2E		
	dValue: DWORD −1	`mov edx, dValue`	EDX, dValue
(d)	AX: 01 4B	`mov ah, 0`	AX
(e)	AL: 64	`mov al, -1`	AL
(f)	EBX: 00 00 3A 4C		
	dValue: DWORD ?	`mov dValue, ebx`	EBX, dValue
(g)	ECX: 00 00 00 00	`mov ecx, 128`	ECX

ASM code for testing

```
TITLE Register Size and Data          (RegSize2.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
.data
;(c)
dValue1   DWORD     -1
;(f)
dValue2   DWORD     ?

.code
main PROC
;(a)
     mov  EBX,0000FF75h
     mov  ECX,000001A2h
     mov  EBX,ECX

;(b)
     mov  EAX,000001A2h
     mov  EAX,100

;(c)
     mov  EDX, 0FF754C2Eh
     mov  EDX,dValue1

;(d)
     mov  EAX,0
     mov  AX,014Bh
     mov  AL,0

;(e)
     mov  EAX,0
     mov  AL,64

;(f)
     mov  EBX,00003A4Ch
     mov  dValue2,EBX

;(g)
     mov  ECX,0
     mov  ECX,128

     call DumpRegs

     exit
main ENDP
END main
```

Debugging for (c)

RegSize2.asm

```
;(b)
      mov   EAX,000001A2h
      mov   EAX,100

;(c)
      mov   EDX, 0FF754C2Eh
      mov   EDX,dValue1

;(d)
      mov   EAX,0
      mov   AX,014Bh
      mov   AL,0

;(e)
      mov   EAX,0
      mov   AL,64

;(f)
      mov   EBX,00003A4Ch
      mov   dValue2,EBX
```

100 %

Memory 1

Address: 0x00401010

```
0x00401010  bb 75 ff 00 00 b9 a2 01 00 0
0x00401020  00 b8 64 00 00 00 ba 2e 4c 7
0x00401030  00 b8 00 00 00 00 66 b8 4b 0
0x00401040  00 b0 40 bb 4c 3a 00 00 89 1
0x00401050  00 00 00 b9 80 00 00 00 e8 e
0x00401060  f0 0f 00 00 cc cc cc cc cc c
0x00401070  cc cc cc cc cc cc cc cc cc c
```

Registers

```
EAX = 00000064 EBX = 000001A2
ECX = 000001A2 EDX = FFFFFFFF
ESI = 00000000 EDI = 00000000
EIP = 00401031 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000246
```

Debugging for (f)

RegSize2.asm

```
;(b)
      mov   EAX,000001A2h
      mov   EAX,100

;(c)
      mov   EDX, 0FF754C2Eh
      mov   EDX,dValue1

;(d)
      mov   EAX,0
      mov   AX,014Bh
      mov   AL,0

;(e)
      mov   EAX,0
      mov   AL,64

;(f)
      mov   EBX,00003A4Ch
      mov   dValue2,EBX
```

100 %

Memory 1

Address: 0x00405000

```
0x00405000  ff ff ff ff 00 00 00 00 00 0
0x00405010  00 30 31 32 33 34 35 36 37 3
0x00405020  46 20 20 20 20 20 20 20 20 2
0x00405030  20 20 20 20 20 20 20 20 20 2
0x00405040  20 20 20 20 20 20 20 20 20 2
0x00405050  20 20 20 20 20 20 20 20 20 2
0x00405060  20 20 20 20 20 20 20 20 20 2
```

Registers

```
EAX = 00000040 EBX = 00003A4C
ECX = 000001A2 EDX = FFFFFFFF
ESI = 00000000 EDI = 00000000
EIP = 00401048 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000246

00405004 = 00000000
```

After (f)

RegSize2.asm

```
;(e)
     mov  EAX,0
     mov  AL,64

;(f)
     mov  EBX,00003A4Ch
     mov  dValue2,EBX

;(g)
     mov  ECX,0
     mov  ECX,128

     call DumpRegs

     exit
main ENDP
END main
```

L. Endian

100 %

Memory 1

Address: 0x00405000

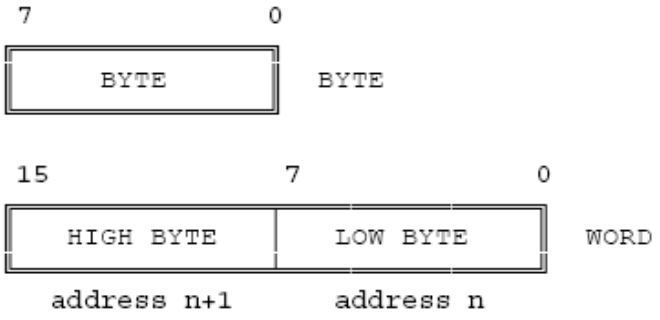
```
0x00405000  ff ff ff ff 4c 3a 00 00 00 0
0x00405010  00 30 31 32 33 34 35 36 37 3
0x00405020  46 20 20 20 20 20 20 20 20 2
0x00405030  20 20 20 20 20 20 20 20 20 2
0x00405040  20 20 20 20 20 20 20 20 20 2
0x00405050  20 20 20 20 20 20 20 20 20 2
0x00405060  20 20 20 20 20 20 20 20 20 2
```

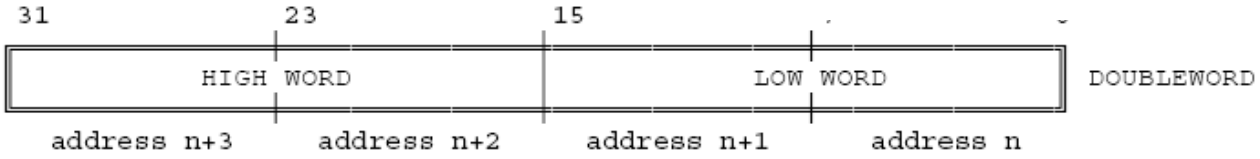
Registers

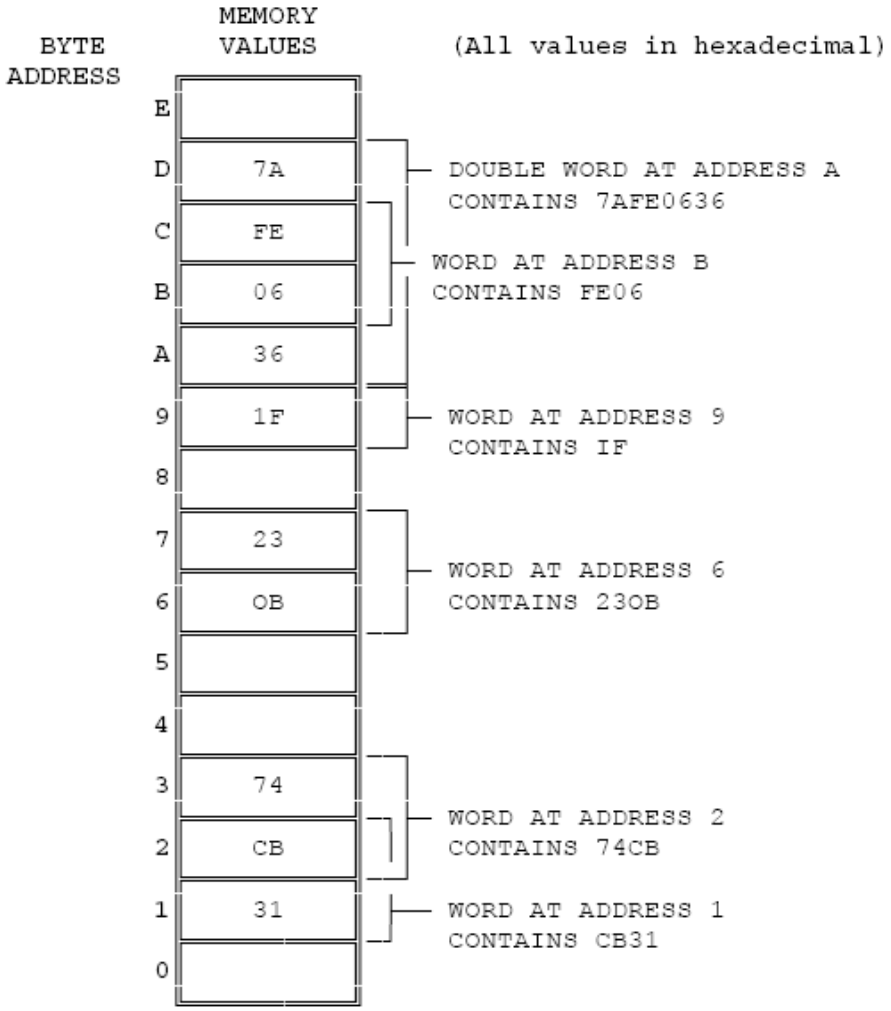
```
EAX = 00000040 EBX = 00003A4C
ECX = 000001A2 EDX = FFFFFFFF
ESI = 00000000 EDI = 00000000
EIP = 0040104E ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000246
```

Basic Data Types

- Byte (BYTE), Words (WORD), Double Words (DWORD)
- Little-Endian
- Align by 2 (word) or 4 (Dword) for better performance – instead of odd address







Data Declaration

- Directives for Data Declaration and Reservation of Memory
  - BYTE: Reserves 1 byte in memory
    - Example:
```
D1   BYTE      20
D2   BYTE      00010100b
String1  BYTE    “Joe” ;
[4A 6F   65]
```
  - WORD: 2 bytes are reserved
    - Example:
```
num1 WORD      -10
num2    WORD      0FFFFh
```
  - DWORD: 4 bytes are reserved
    - Example:
```
N1        DWORD     -10
```
  - QWORD: 8 bytes
    - 64 bit: RAX RBX RCX ,etc
    - 32 bit: **EDX:EAX** Concatenation for **CDQ** instruction

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Instruction Format

- Opcode:
  - specifies the operation performed by the instruction.
- Register specifier
  - an instruction may specify one or two register operands.
- Addressing-mode specifier
  - when present, specifies whether an operand is a register or memory location.
- Displacement
  - when the addressing-mode specifier indicates that a displacement will be used to compute the address of an operand, the displacement is encoded in the instruction.
- Immediate operand
  - when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide.

```
mov   eax, source
mov   dest, eax
mov   eax, source+4
mov   dest+4, eax
mov   eax, source+8
mov   dest+8, eax
mov   eax, source+12
mov   dest+12, eax
```

```
mov   eax, 0
```

386 Instruction Set

- 9 Operation Categories
  - Data Transfer
  - Arithmetic
  - Shift/Rotate
  - String Manipulation
  - Bit Manipulation
  - Control Transfer
  - High Level Language Support
  - Operating System Support
  - Processor Control
- Number of operands: 0, 1, 2, or 3

Table 2-2b. Arithmetic Instructions

ADDITION	
ADD	Add operands
ADC	Add with carry
INC	Increment operand by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
SUBTRACTION	
SUB	Subtract operands
SBB	Subtract with borrow
DEC	Decrement operand by 1
NEG	Negate operand
CMP	Compare operands
DAS	Decimal adjust for subtraction
AAS	ASCII Adjust for subtraction
MULTIPLICATION	
MUL	Multiply Double/Single Precision
IMUL	Integer multiply
AAM	ASCII adjust after multiply
DIVISION	
DIV	Divide unsigned
IDIV	Integer Divide
AAD	ASCII adjust before division

Data movement Instructions

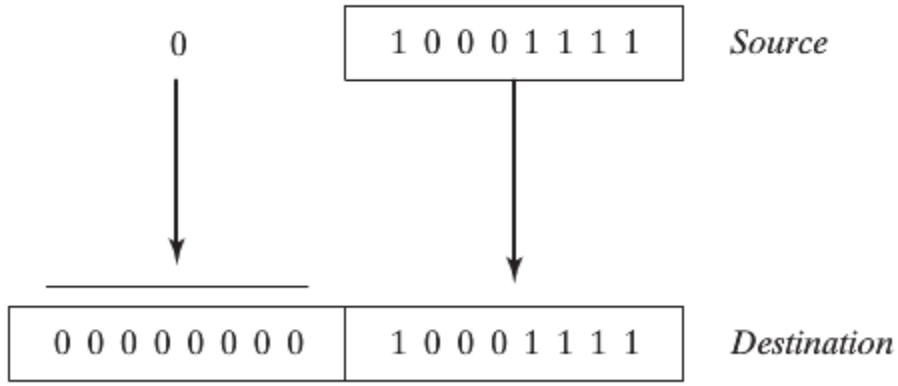
- MOV (Move)
  - transfers a byte, word, or doubleword from the source operand to the destination operand: R→ M, M → R, R→ R, I→R, I→ M
  - The MOV instruction cannot move M→M
  - M→M via MOVS (string)
- MOVZX (Move with Zero-Extended)
- MOVSX (Move with Sign-Extended)
- XCHG (Exchange)
  - swaps the contents of two operands.
  - swap two byte operands, two word operands, or two doubleword operands.
  - The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand.

MOVZX and MOVSX

- MOVZX

mov AL, 8Fh
movzx AX, AL

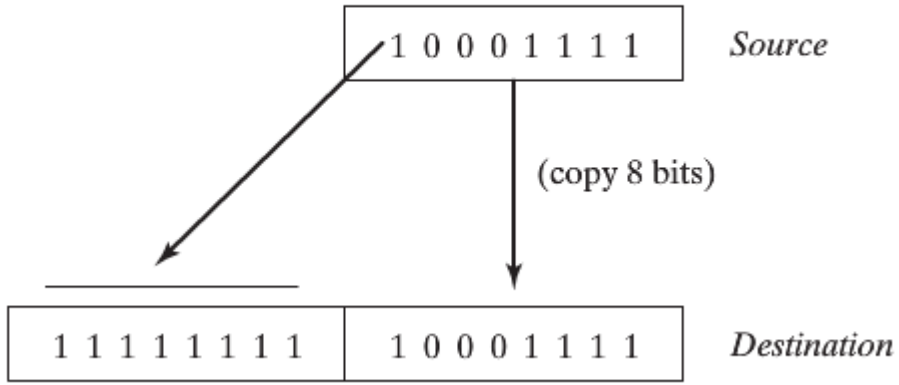
Using MOVZX to copy a byte into a 16-bit destination.



- MOVSX

mov AL, 8Fh
movsx, AX, AL

Using MOVSX to copy a byte into a 16-bit destination.



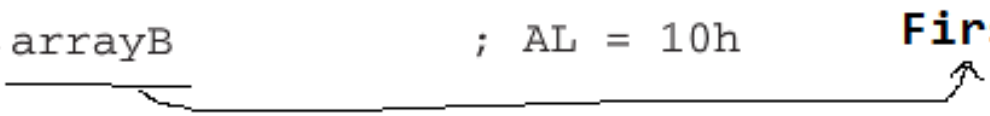
Direct-Offset Operands

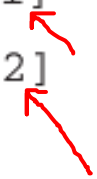
- Add displacement to the name of a variable
- Accessing memory locations that may not have explicit labels
- BYTE Case [AL]

```
arrayB  BYTE 10h,20h,30h,40h,50h

mov  al,arrayB              ; AL = 10h

mov  al,[arrayB+1]          ; AL = 20h
mov  al,[arrayB+2]          ; AL = 30h
```





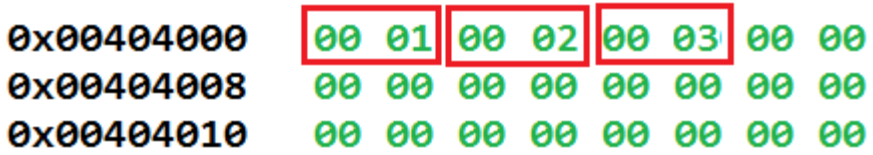
```
0x00404000   10 20 30 40 50 00 00 00
0x00404008   00 00 00 00 00 00 00 00
0x00404010   00 00 00 00 00 00 00 00
```

Direct-Offset Operands

- WORD case [AX]

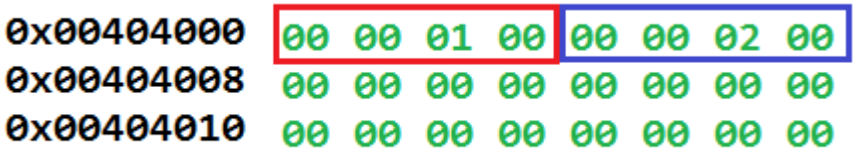
```
.data
arrayW WORD 100h,200h,300h

.code
mov  ax,arrayW          ; AX = 100h
mov  ax,[arrayW+2]      ; AX = 200h
```



- DWORD case [EAX]

```
.data
arrayD DWORD 10000h,20000h
.code
mov  eax,arrayD         ; EAX = 10000h
mov  eax,[arrayD+4]     ; EAX = 20000h
```



Example Code /ch04/moves.asm

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

; Chapter 4 example. Demonstration of MOV and
; XCHG with direct and direct-offset operands.

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

;  MOVSX
     mov   bx,0A69Bh
     movsx eax,bx           ; EAX = FFFFA69Bh
     movsx edx,bl           ; EDX = FFFFFF9Bh
     mov   bl,7Bh
     movsx cx,bl            ; CX  = 007Bh

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

;  Direct-Offset Addressing (byte array):
     mov al,arrayB          ; AL = 10h
     mov al,[arrayB+1]      ; AL = 20h
     mov al,[arrayB+2]      ; AL = 30h

;  Direct-Offset Addressing (word array):
     mov ax,arrayW          ; AX = 100h
     mov ax,[arrayW+2]      ; AX = 200h

;  Direct-Offset Addressing (doubleword array):
     mov eax,arrayD                 ; EAX = 10000h
     mov eax,[arrayD+4]             ; EAX = 20000h

     exit

main ENDP
END main
```

Data and Code Segment

```
; Chapter 4 example. Demonstration of MOV and
; XCHG with direct and direct-offset operands.

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

100 %

Memory 1

Address: 0x00404000

```
0x00404000  00 10 00 20 10 20 30 40  ...
0x00404008  50 00 01 00 02 00 03 00  P..
0x00404010  00 01 00 00 00 02 00 00  ...
0x00404018  00 00 00 00 00 00 00 00  ...
0x00404020  00 00 00 00 00 00 00 00  ...
0x00404028  00 00 00 00 00 00 00 00  ...
0x00404030  00 00 00 00 00 00 00 00  ...
0x00404038  00 00 00 00 00 00 00 00
```

Memory 2

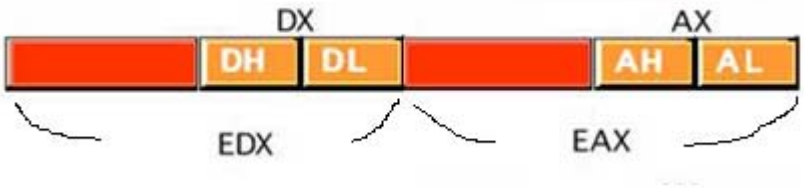
Address: 0x00401000

```
0x00401000  cc cc cc cc cc e9 06  ÌÌÌÌÌé.
0x00401007  00 00 00 cc cc cc cc  ...ÌÌÌÌ
0x0040100E  cc cc 66 bb 9b a6 0f  ÌÌf».¦.
0x00401015  b7 c3 0f b6 d3 66 0f  ·Ã.¶Óf.
0x0040101C  b6 cb 66 bb 9b a6 0f  ¶Ëf».¦.
0x00401023  bf c3 0f be d3 b3 7b  ¿Ã..Ó.{
```

Memory 2 | Registers

Data type Conversion Instructions

- CBW (Convert Byte to Word)
  - **extends the sign of the byte in register AL throughout AX.**
- CWDE (Convert Word to Doubleword Extended)
  - **extends the sign of the word in register AX throughout EAX**.
- CWD (Convert Word to Doubleword)
  - **extends the sign of the word in register AX throughout register DX**
  - can be used to produce a doubleword dividend from a word before a word division
- CDQ (Convert Doubleword to Quad-Word)
  - **extends the sign of the doubleword in EAX throughout EDX.**
  - can be used to produce a quad-word dividend from a doubleword before doubleword division.



Data type Conversion Instructions – Practice

- CBW (Convert Byte to Word)
  - **extends the sign of the byte in register AL throughout AX.**
- CWDE (Convert Word to Doubleword Extended)
  - **extends the sign of the word in register AX throughout EAX**.
- CWD (Convert Word to Doubleword)
  - **extends the sign of the word in register AX throughout register DX**
  - can be used to produce a doubleword dividend from a word before a word division
- CDQ (Convert Doubleword to Quad-Word)
  - **extends the sign of the doubleword in EAX throughout EDX.**
  - can be used to produce a quad-word dividend from a doubleword before doubleword division.

```
MOV EAX,12345678h
MOV EDX,11111111h
MOV AL,8Fh
CBW          ;Byte to Word
             ;EAX= [              ]
CWDE         ;WORD to DWORD
             ;EAX = [             ]

CWD          ;WORD to DWORD
             ;  EAX= [            ]
             ;  EDX= [            ]
CDQ          ;DWORD to QWORD
             ;  EAX = [           ]
             ;  EDX = [            ]
```

Data type Conversion Instructions – Code

Conversion.asm

```
TITLE Data Type Conversion Examples       (Conversion.asm)


INCLUDE Irvine32.inc

.code
main PROC

      mov  EAX,12345678h
      mov  EDX,76543210h
      mov  AL, 8Fh
      cbw
      cwde
      cwd
      cdq

      exit

main ENDP
END main
```

100 %

Memory 1

Address: 0x00404000

```
0x00404000  28 40 00 00 00 00 00 00 00 00
0x0040401D  00 00 00 00 00 00 00 00 00 00
0x0040403A  00 00 00 00 00 00 00 00 00 00
0x00404057  00 d8 79 d3 75 00 00 00 00 00
0x00404074  00 00 00 00 00 00 00 00 00 00
0x00404091  63 65 73 73 00 4b 45 52 4e 45
0x004040AE  00 00 00 00 00 00 00 00 00 00
```

Registers

```
EAX = FFFFFF8F EBX = 7EFDE000
ECX = 00000000 EDX = FFFFFFFF
ESI = 00000000 EDI = 00000000
EIP = 00401022 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000246
```

Memory 2 | Registers

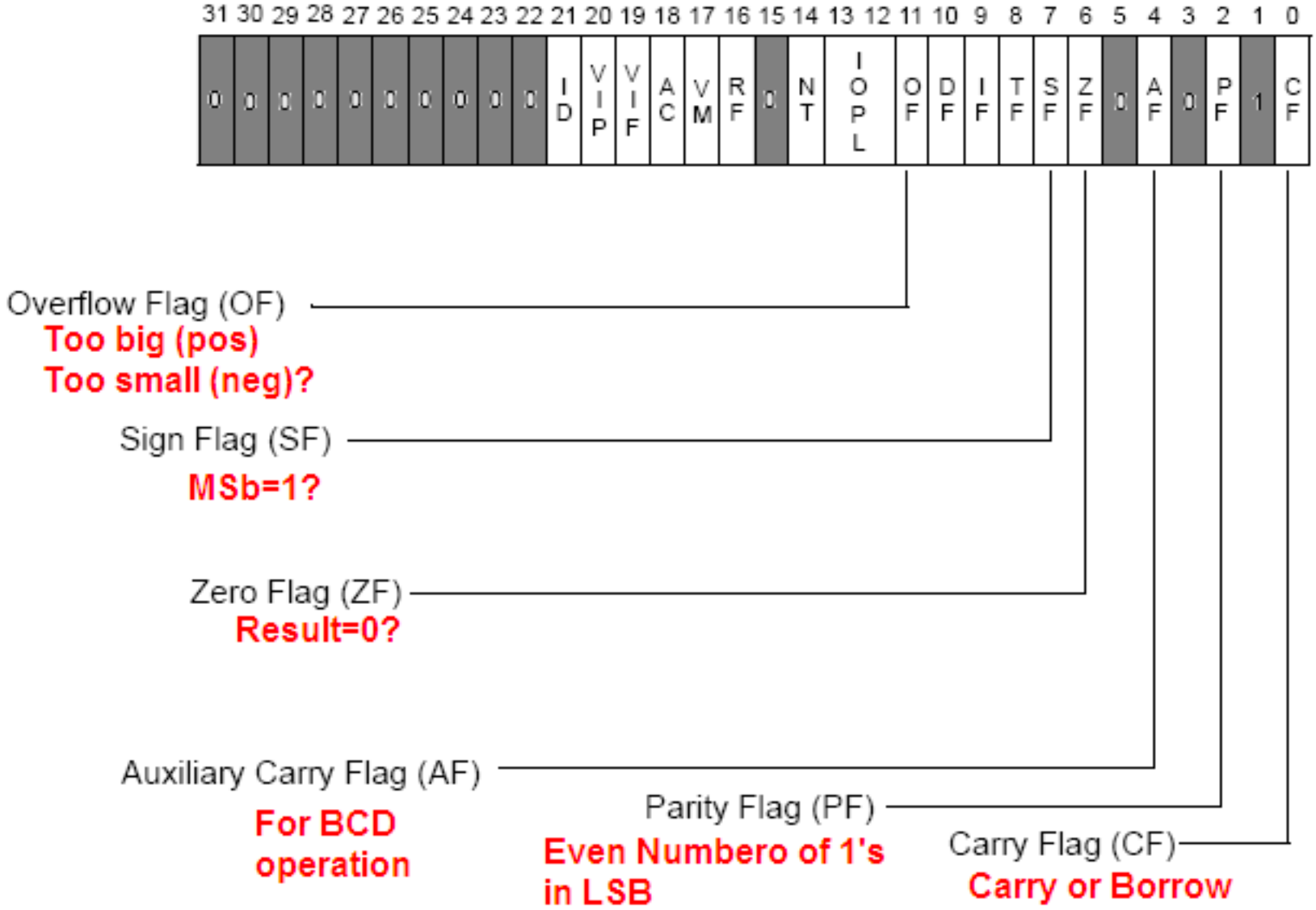
- **CBW (Convert Byte to Word)**
  - extends the sign of the byte in register AL throughout AX.
- **CWDE (Convert Word to Doubleword Extended)**
  - extends the sign of the word in register AX throughout EAX.
- **CWD (Convert Word to Doubleword)**
  - extends the sign of the word in register AX throughout register DX
  - can be used to produce a doubleword dividend from a word before a word division
- **CDQ (Convert Doubleword to Quad-Word)**
  - extends the sign of the doubleword in EAX throughout EDX.
  - can be used to produce a quad-word dividend from a doubleword before doubleword division.

Addition Instruction

- ADD (Add Integers)
  - (DST + SRC) → DST
  - replaces the destination operand with the sum of the source and destination operands. OF, SF, ZF, CF are all affected.

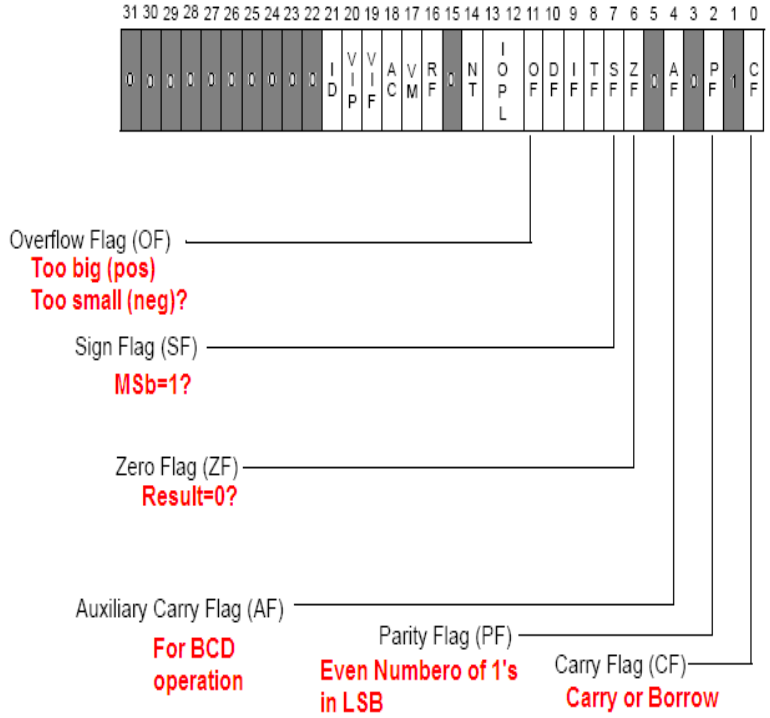
Before	*Instruction Executed*	*After*				
EAX: 00 00 00 75 ECX: 00 00 01 A2	`add eax, ecx`	EAX	00	00	02	17
		ECX	00	00	01	A2
		SF 0 ZF 0 CF 0 OF 0				

Status Flags



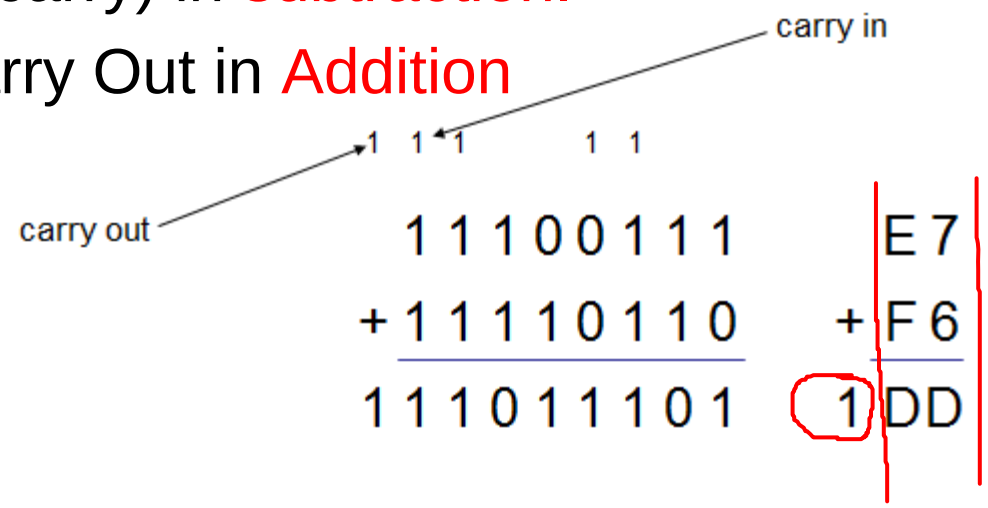
Status Flags

- CF (Carry Flag) EFL[0]
  - 1: Result of unsigned operation is too large
  - 0: otherwise
- PF (Parity Flag) EFL[2]
  - 1: LSB contains an even number of 1's
  - 0: odd number of 1's
- AF (Auxiliary Carry Flag) EFL[4]
  - 1: Carry from bit 3 to bit 4 in an 8-bit operation
  - 0: Otherwise
- ZF (Zero Flag) EFL[6]
  - 1: Result is zero (0)
  - 0: Non-zero
- SF (Sign Flag) EFL[7]
  - 1: Result is Negative
  - 0: Positive
- OF (Overflow Flag) EFL[11]
  - 1: Result of signed operation is too large
  - 0: Otherwise



Flags: CF, ZF, SF

- **SF (Sign Flag):** 1 (neg) 0 (pos)
- **ZF (Zero Flag):** 1 (result is zero) 0 (otherwise)
- **CF (Carry Flag)**
  - If the sum of two numbers is one bit longer than the operands, the extra 1 is a carry (or **carry out**) → CF=1
    - A 1 carried into the **highest-order (sign, leftmost) bit position** during addition is called a "**carry in**".
  - CF=1 for borrow (or no carry) in subtraction.
  - CF =1 when there is Carry Out in Addition



Flags1.asm

LAHF Load Flags into AH Register

Transfers bits 0 to 7 of the flags register to AH.

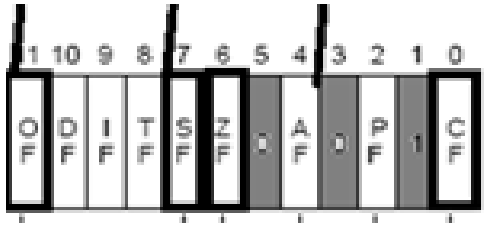
```
.code
main PROC

;  (1)
     mov  EAX,0
     mov  EBX,0
     mov  AL, 0E7h
     mov  BL, 0F6h
     add  BL,AL        ; BL = E7 + F6
     lahf              ; Load the Flags bits 7 - 0 to AH reg
     call DumpRegs     ; Show the contents of Reg and Flag

; (2)
     mov  EAX,0
     mov  EBX,0
     mov  AX, 483Fh
```

C:\Irvine\Examples\ch04\Project\Debug\Project.exe

```
EAX=000087E7  EBX=000000DD  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401026  EFL=00000287  CF=1  SF=1  ZF=0  OF=0  AF=0  PF=1
```



0 %

Memory 1

Address: 0x00401000

```
00401000  cc cc cc cc cc e9 06 00 00 00 cc cc cc cc cc c
0040101D  f6 02 d8 9f e8 1e 02 00 00 b8 00 00 00 00 bb 0
0040103A  c3 e8 04 02 00 00 b8 00 00 00 00 bb 00 00 00 0
00401057  00 00 6a 00 e8 f4 0f 00 00 cc cc cc cc cc cc c
```

Registers

```
EAX = 000087E7 EBX = 000000DD
ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 00401026 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000287
```

Flags: OF

- **OF (Overflow flag)**
  - OF=1 when there is a CARRY IN but no CARRY OUT
  - OF=1 when there is a CARRY OUT but no CARRY IN
  - If OF=1, result is wrong when adding 2 signed numbers

- Example

  483F + 645A → AC99

  Carry In but no Carry Out

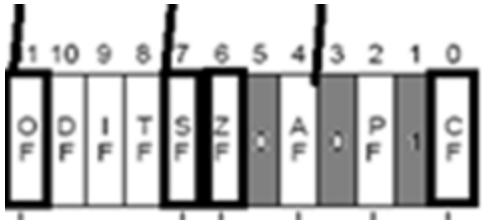
  → OF=1

  No Carry Out → CF=0

```
1 Carry In
↖
0100 1000 0011 1111
0110 0100 0101 1010  +
--------------------
1010 1100 1001 1001
```

- Interpretation:
  - If the operation is for unsigned number addition → Correct
  - If the operation is for signed numbers → Incorrect

Flags1.asm




```
; (2)
      mov  EAX,0
      mov  EBX,0
      mov  AX, 483Fh
      mov  BX, 645Ah
      add  AX,BX      ; AX = 483F + 645A
      call DumpRegs
; (3)
      mov  EAX,0
      mov  EBX,0
      mov  EAX, 0FFFFFF97h
      add  EAX, 158   ; EAX = EAX + 158
      call DumpRegs

      exit

main ENDP
END main
```

C:\Irvine\Examples\ch04\Project\Debug\Project.exe

```
EAX=000087E7  EBX=000000DD  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401026  EFL=00000287  CF=1  SF=1  ZF=0  OF=0  AF=0  PF=1

EAX=0000AC99  EBX=0000645A  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401040  EFL=00000A96  CF=0  SF=1  ZF=0  OF=1  AF=1  PF=1
```

%

mory 1

dress: 0x00401000

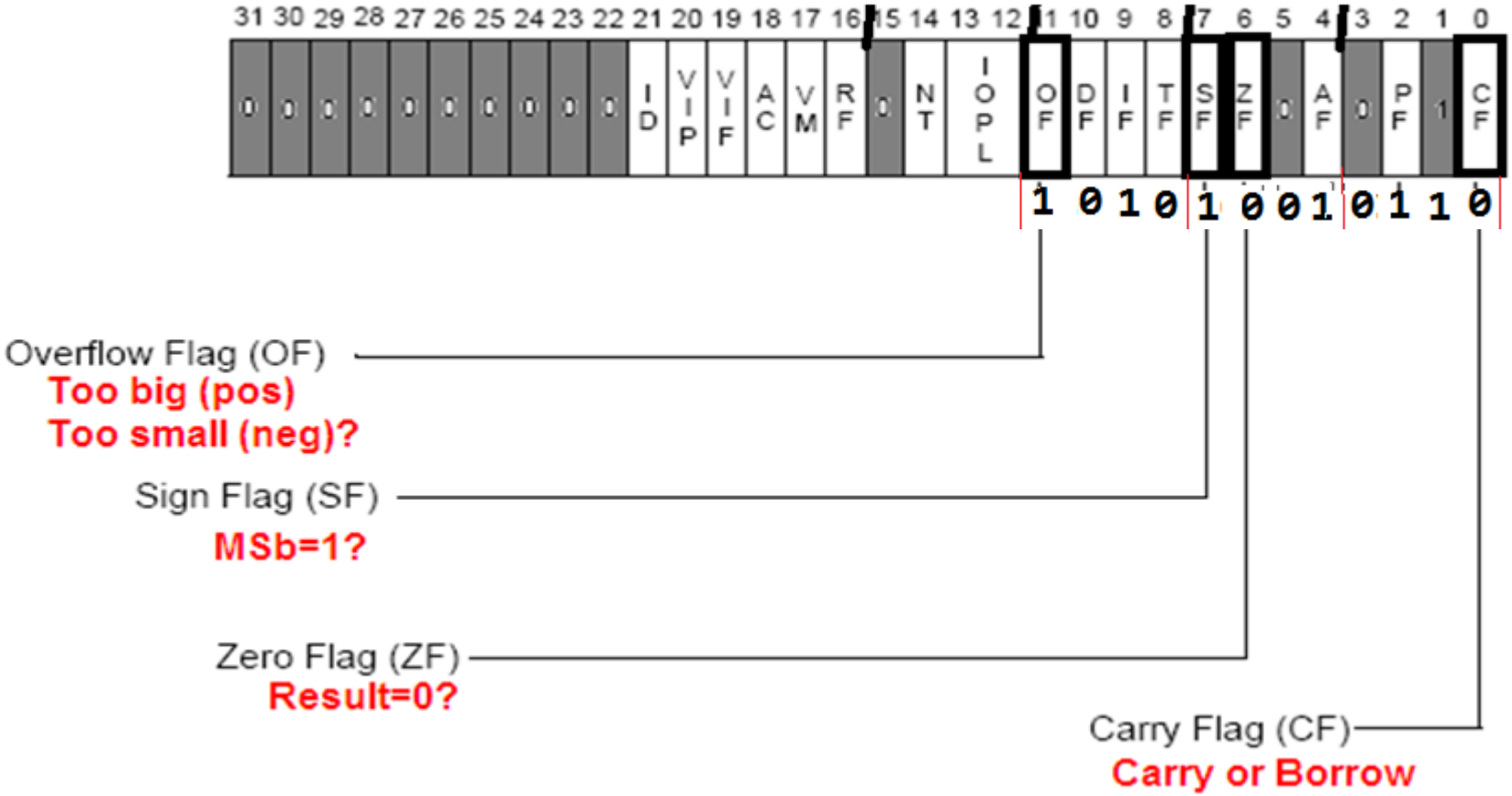
```
00401000  cc cc cc cc cc e9 06 00 00 00 cc cc cc cc cc c
0040101D  f6 02 d8 9f e8 1e 02 00 00 b8 00 00 00 00 bb 0
0040103A  c3 e8 04 02 00 00 b8 00 00 00 00 bb 00 00 00 0
00401057  00 00 6a 00 e8 f4 0f 00 00 cc cc cc cc cc cc c
00401074  cc cc cc cc cc cc cc cc cc cc cc cc 50 e8 d4 0
```

Registers

```
EAX = 0000AC99 EBX = 0000645A
ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 00401040 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000A96
```

Status Flags --- BEFORE

- BEFORE:
  - EAX= FFFFFF97h
  - EFL= 00000A96h

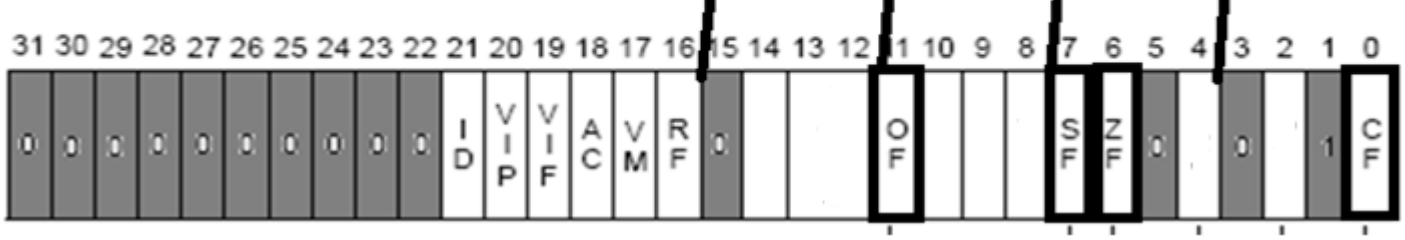


Status Flags -- AFTER

- AFTER: **`add eax, 158`**
  - EAX= 00000035h
  - EFL= 00000217h

```
                 1 1
FF FF FF 97   1111 1111 1111 1111 1111 1111 1001 0111
          9E  0000 0000 0000 0000 0000 0000 1001 1110
-----------  --------------------------------------
00 00 00 35  0000 0000 0000 0000 0000 0000 0011 0101

             0000 0000 0000 0000 0000 0010 0001 0111
```



Flags1.asm

```
; (2)
    mov  EAX,0
    mov  EBX,0
    mov  AX, 483Fh
    mov  BX, 645Ah
    add  AX,BX      ; AX = 483F + 645A
    call DumpRegs
; (3)
    mov  EAX,0
    mov  EBX,0
    mov  EAX, 0FFFFFF97h
    add  EAX, 158   ; EAX = EAX + 158
    call DumpRegs

    exit

main ENDP
END main
```

C:\Irvine\Examples\ch04\Project\Debug\Project.exe

```
EAX=000087E7  EBX=000000DD  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401026  EFL=00000287  CF=1  SF=1  ZF=0  OF=0  AF=0  PF=1

EAX=0000AC99  EBX=0000645A  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401040  EFL=00000A96  CF=0  SF=1  ZF=0  OF=1  AF=1  PF=1

EAX=00000035  EBX=00000000  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401059  EFL=00000217  CF=1  SF=0  ZF=0  OF=0  AF=1  PF=1
```

Memory 1

Address: 0x00401000

```
00401000  cc cc cc cc cc e9 06 00 00 00 cc cc cc cc cc c
0040101D  f6 02 d8 9f e8 1e 02 00 00 b8 00 00 00 00 bb 0
0040103A  c3 e8 04 02 00 00 b8 00 00 00 00 bb 00 00 00 0
00401057  00 00 6a 00 e8 f4 0f 00 00 cc cc cc cc cc cc c
00401074  cc cc cc cc cc cc cc cc cc cc cc cc 50 e8 d4 0
00401091  50 40 00 00 75 05 e8 ff 04 00 00 8d 45 ea 50 f
004010AE  ea 66 a3 11 56 40 00 81 3d 11 56 40 00 00 02 0
```

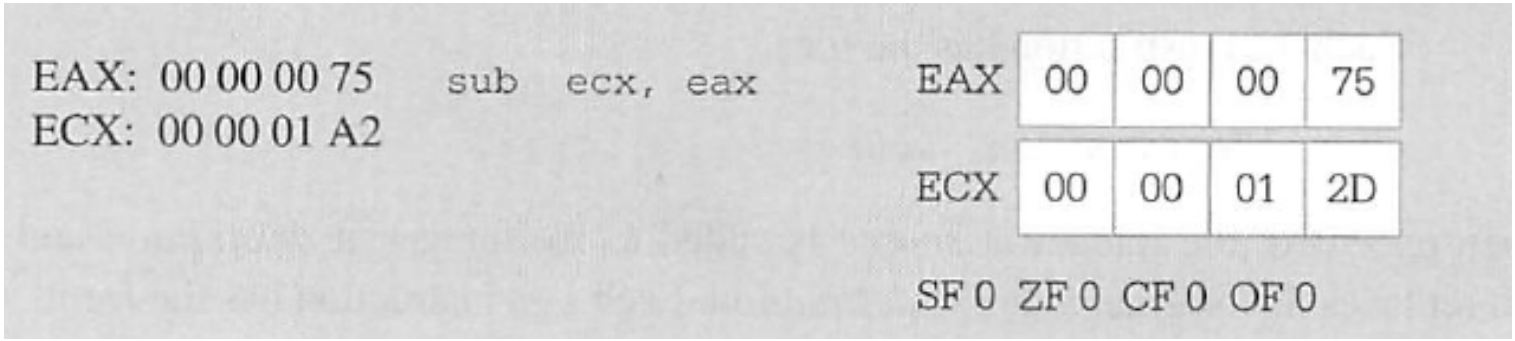
Registers

```
EAX = 00000035 EBX = 00000000
ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 00401059 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000217
```

SUB (Subtract Integers)

- **SUB:**
  - Operation: (DST – SRC) → DST
  - subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required**, the CF is set.** The operands may be signed or unsigned bytes, words, or doublewords.

- label mnemonic dst, src



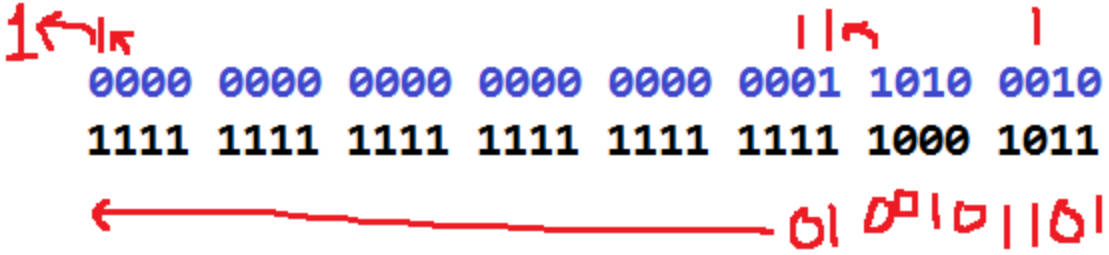
SUB (Subtract Integers) – Manual Check

```
   ECX -   EAX

   ECX     00 00 01 A2
-) EAX     00 00 00 75
```

```
                                ECX     00 00 01 A2
          +) 16's Complement of EAX     FF FF FF 8B
                                      ---------------
                                      00 00 01 2D
```

For Flag Check --> Binary



Cary In & Carry Out --> OF=0

CF=0 because Carry Out (*Note - subtraction)

ADD & SUB Examples --- Manual Check

```
EAX: 00 00 00 75      sub  eax, ecx
ECX: 00 00 01 A2
```

EAX	FF	FF	FE	D3
ECX	00	00	01	A2

SF 1 ZF 0 CF 0 OF 0

```
AX: 77 AC             add  ax, cx
CX: 4B 35
```

AX	C2	E1
CX	4B	35

SF 1 ZF 0 CF 0 OF 1

```
EAX: 00 00 00 75      sub  ecx, eax
ECX: 00 00 01 A2
```

EAX	00	00	00	75
ECX	00	00	01	2D

SF 0 ZF 0 CF 0 OF 0

```
BL: 4B                add  bl, 4
```

BL	4F

SF 0 ZF 0 CF 0 OF 0

- SUB [dst] – [src]

SF: Sign Falg
ZF: Zero Flag
CF: Carry Flag
OF: Overflow Flag

```
DX: FF 20             sub  dx, Value
word at value: FF 20
```

DX	00	00
Value	FF	20

SF 0 ZF 1 CF 0 OF 0

```
EAX: 00 00 00 09      add  eax, 1
```

EAX	00	00	00	0A

SF 0 ZF 0 CF 0 OF 0

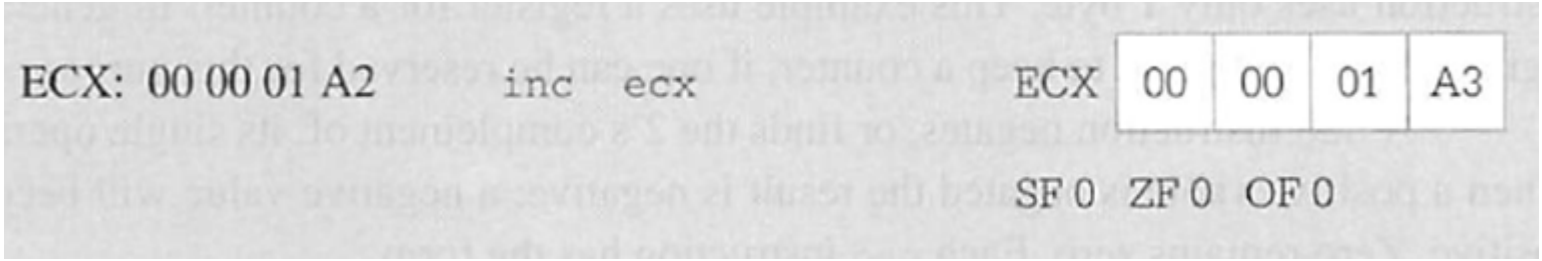
```
doubleword at Dbl:    sub  Dbl, 1
   00 00 01 00
```

Dbl	00	00	00	FF

SF 0 ZF 0 CF 0 OF 0

INC & DEC

- INC (Increment)
  - DST +1 → DST
  - adds one to the destination operand. **INC does not affect CF.** Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed.



- DEC (Decrement)
  - DST – 1 → DST
  - subtracts 1 from the destination operand. **DEC does not update CF.** Use SUB with an immediate value of 1 to perform a decrement that affects carry.

INC + DEC examples

Example

Before	*Instruction executed*	*After*
ECX: 00 00 01 A2	`inc ecx`	ECX 00 00 01 A3 SF 0 ZF 0 OF 0
AL: F5	`dec al`	AL F4 SF 1 ZF 0 OF 0
word at Count: 00 09	`inc Count`	Count 00 0A SF 0 ZF 0 OF 0
BX: 00 01	`dec bx`	BX 00 00 SF 0 ZF 1 OF 0
EDX: 7F FF FF FF	`inc edx`	EDX 80 00 00 00 SF 1 ZF 0 OF 1

CMP + NEG

- CMP (Compare)
  - **DST – SRC**
  - **subtracts** the source operand from the destination operand. **It updates OF, SF, ZF, AF, PF, and CF** but **does not alter the source and destination operands.**

```
cmp    eax, 356
cmp    wordOp, 0d3a6h
cmp    bh, '$'
```

- NEG (Negate)
  - **0 – DST → DST**
  - **subtracts a signed integer operand from zero**. The effect of NEG is to **reverse the sign** of the operand from positive to negative or from negative to positive (i.e., **16's complement**)
  - **SF and ZF are affected**

NEG Examples

Example

Before	*Instruction executed*	*After*
BX: 01 A2	`neg bx`	BX FE 5E SF 1 ZF 0
DH: F5	`neg dh`	DH 0B SF 0 ZF 0
word at Flag: 00 01	`neg Flag`	Flag FF FF SF 1 ZF 0
EAX: 00 00 00 00	`neg eax`	EAX 00 00 00 00 SF 0 ZF 1

Inc/Neg Practice

IncNeg.asm

```
INCLUDE Irvine32.inc
.data
prompt    BYTE "Enter your number:  ",0
x     DWORD     ?
y     DWORD     ?
z     DWORD     ?

.code
main PROC
      mov  EAX,0
      mov  EBX,0

      mov  EDX,OFFSET prompt
      call WriteString

      call ReadDec
;Others -- ReadHex (Hex number), ReadInt (signed number)
      mov  x,EAX
```

```
      mov  EDX,OFFSET prompt
      call WriteString

      call ReadDec
      mov  z, EAX

      mov  EAX, x
      add  EAX, y
      mov  EBX,z
      add  EBX,EBX
      call DumpRegs
      sub  EAX,EBX
      call DumpRegs
      inc  EAX
      call DumpRegs
      neg  EAX
      call DumpRegs

      exit
main ENDP
```

100 %

Memory 1

Address: 0x00405000

```
0x00405000  45 6e 74 65 72 20 79 6f  Enter yo
0x00405008  75 72 20 6e 75 6d 62 65  ur numbe
0x00405010  72 3a 20 20 00 23 00 00  r:  .#..
0x00405018  00 2f 00 00 00 1a 00 00  ./......
0x00405020  00 00 00 00 00 00 00 00  ........
```

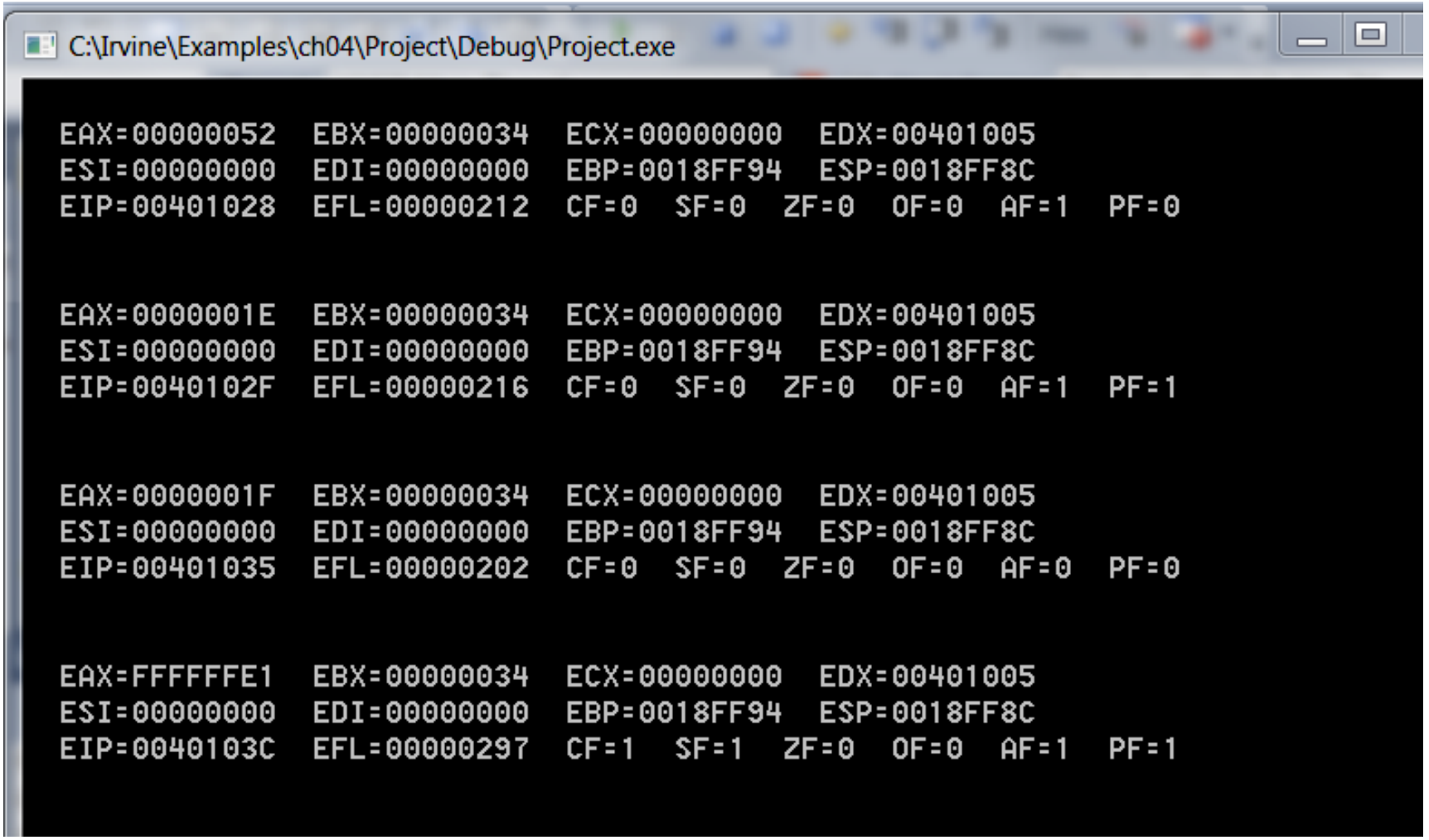
Registers

```
EAX = 00000052 EBX = 00000034
ECX = 00000000 EDX = 00405000
ESI = 00000000 EDI = 00000000
EIP = 0040106E ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000212
```

Link Library Procedures – Just a few

- DumpRegs
  - Displays EAX, EBX, etc
- ReadDec
  - Reads a 32-bit unsigned decimal integer from keyboard and returns the value in EAX
- ReadHex
  - Reads a 32-bit unsigned hex integer from the keyboard and returns the value in EAX
- ReadInt
  - Reads a 32-bit signed decimal integer from the keyboard and returns the value in EAX
- WriteString
  - Write a null-terminated string to the console window (pass the string's offset in EDX)

DumpRegs


C:\Irvine\Examples\ch04\Project\Debug\Project.exe
EAX=00000052 EBX=00000034 ECX=00000000 EDX=00401005
ESI=00000000 EDI=00000000 EBP=0018FF94 ESP=0018FF8C
EIP=00401028 EFL=00000212 CF=0 SF=0 ZF=0 OF=0 AF=1 PF=0
EAX=0000001E EBX=00000034 ECX=00000000 EDX=00401005
ESI=00000000 EDI=00000000 EBP=0018FF94 ESP=0018FF8C
EIP=0040102F EFL=00000216 CF=0 SF=0 ZF=0 OF=0 AF=1 PF=1
EAX=0000001F EBX=00000034 ECX=00000000 EDX=00401005
ESI=00000000 EDI=00000000 EBP=0018FF94 ESP=0018FF8C
EIP=00401035 EFL=00000202 CF=0 SF=0 ZF=0 OF=0 AF=0 PF=0
EAX=FFFFFFE1 EBX=00000034 ECX=00000000 EDX=00401005
ESI=00000000 EDI=00000000 EBP=0018FF94 ESP=0018FF8C
EIP=0040103C EFL=00000297 CF=1 SF=1 ZF=0 OF=0 AF=1 PF=1

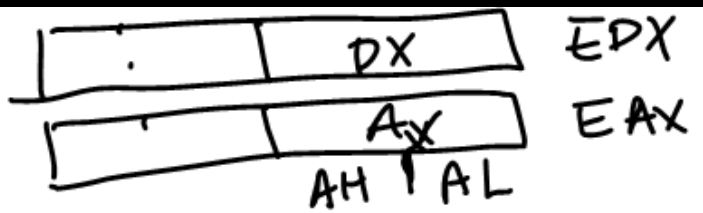
Multiplication Instruction - MUL

Syntax:

MUL *Source* → E(A?) x Source (L X)

- MUL (Unsigned Integer Multiply)
  - performs an unsigned multiplication of the source operand and the accumulator [(E)AX].
  - **If the source is a byte**, the processor multiplies it by the contents of AL and returns the double-length result to AH and AL (Concatenated) i.e, AX.
  - **If the source operand is a word**, the processor multiplies it by the contents of AX and returns the double-length result to DX and AX.
  - **If the source operand is a double-word**, the processor multiplies it by the contents of EAX and returns the 64-bit result in EDX and EAX (Concatenated). MUL sets CF and OF when the upper half of the result is nonzero; otherwise, they are cleared.
  - Operand **cannot** be immediate

MUL Opr/Store Summary



MUL byte

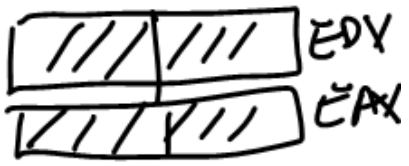
opr → [AL] x [byte] → Store [AX]

MUL BX ← word

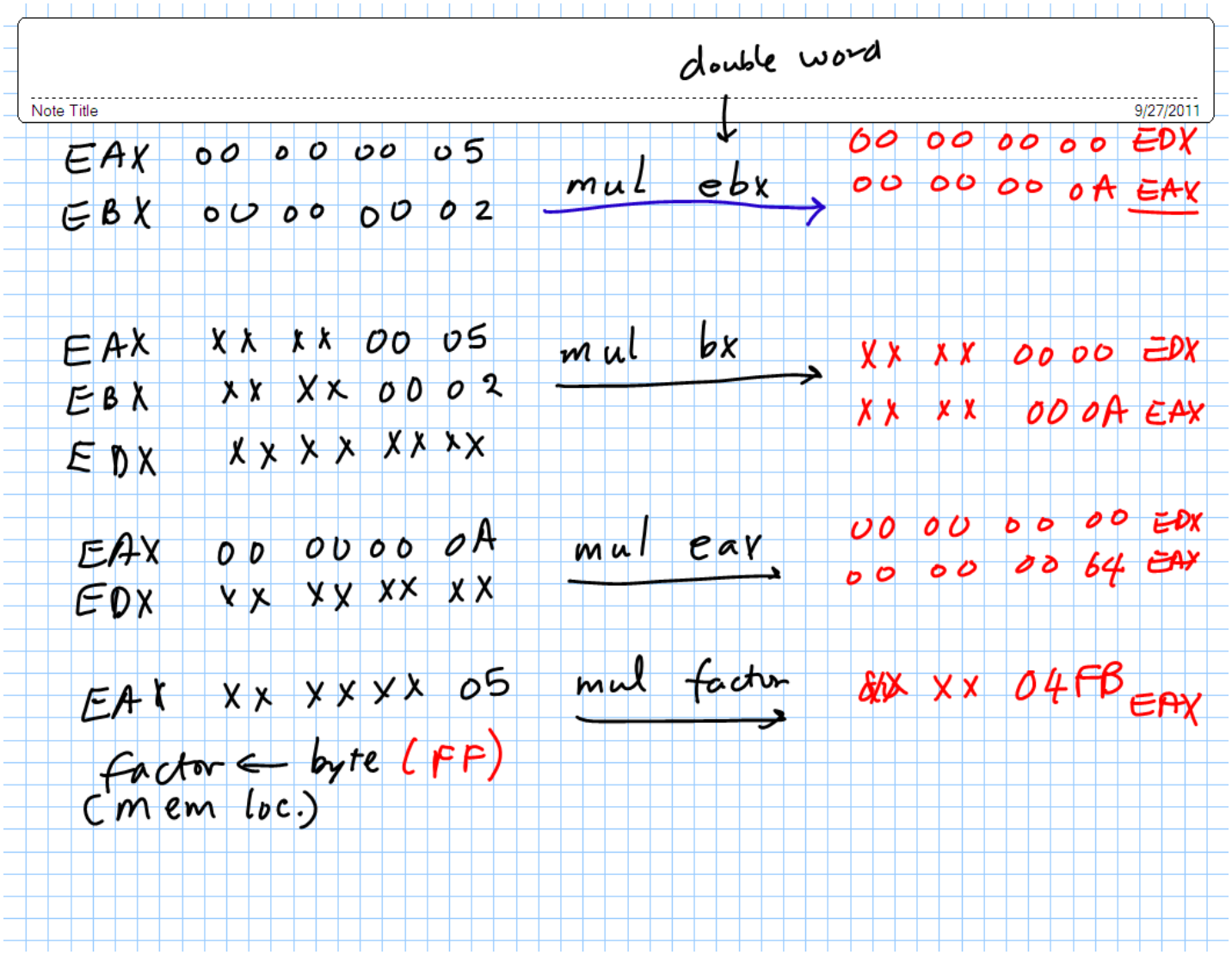
opr → [AX] x [BX] → Store [DX | AX]

MUL EBX ← D-word

opr → [EAX] x [EBX] → Store [EDX | EAX]



MUL - Exercise



IMUL (Signed Integer Multiply)

- performs a signed multiplication operation. IMUL has three variations:
  - 1. An **one-operand form.** The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.

```
imul    source
```

  - 2**. A two-operand form. One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.**

```
imul    destination register, source
```

  - **The immediate operand is treated as signed**. If the immediate operand is **a byte, the processor automatically sign-extends to the size of destination before performing the multiplication.**

IMUL

EAX 00 00 00 05
EBX 00 00 00 02
imul ebx →
EDX 00 00 00 00
EAX 00 00 00 0A

EAX XX XX 00 05
EBX XX XX 00 02
imul bx →
EDX XX XX 00 00
EAX XX XX 00 0A

EAX XX XX XX 05
factor (FF)
imul factor →
EAX XX XX FF FB

EBX 00 00 00 0A
imul ebx, 10 →
EBX 00 00 00 64

AX: ?? 05
byte at *factor*: FF

-1
imul factor → -5
-5

AX | FF | FB

CF, OF 0

Division Instruction

Q R

Syntax:

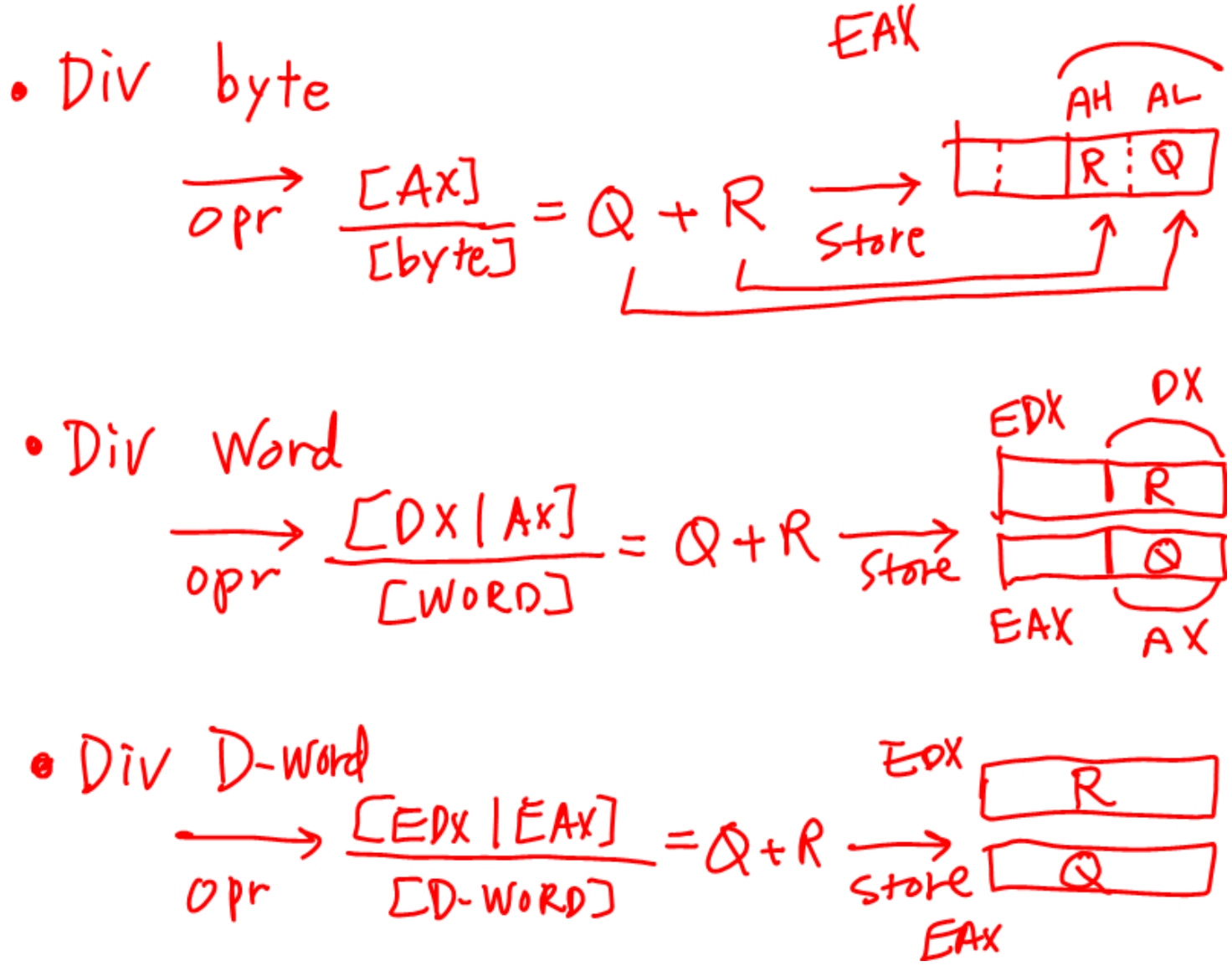
DIV Divider → E(A?)/Divider

- DIV (Unsigned Integer Divide)
  - performs an unsigned division of the accumulator by the source operand.
  - The dividend (the accumulator) is twice the size of the divisor (the source operand)
  -

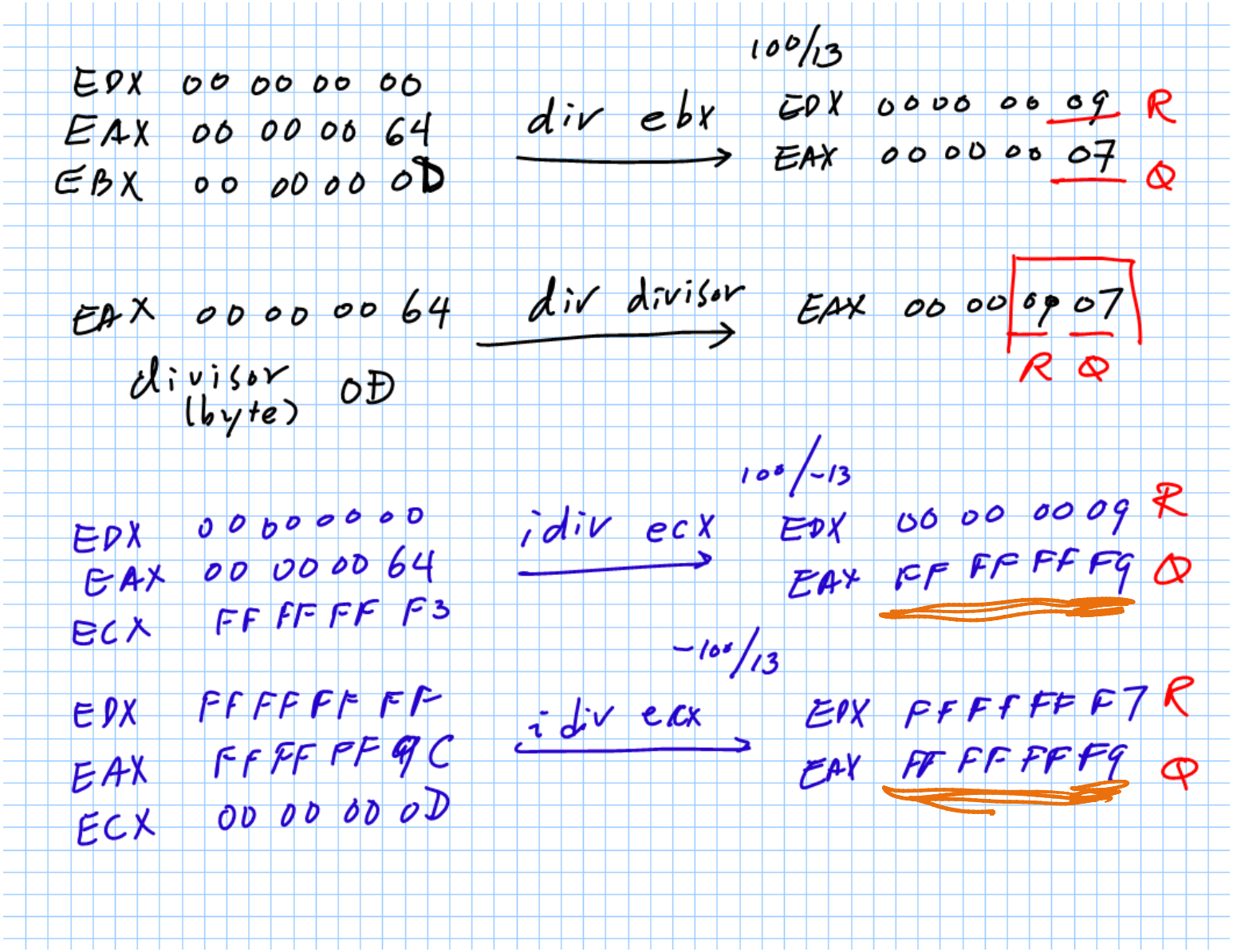
Size of Source Operand (divisor)	Dividend	Quotient	Remainder
Byte	AX	AL	AH
Word	DX:AX	AX	DX
Doubleword	EDX:EAX	EAX	EDX

- IDIV (Signed Integer Divide)
  - performs a signed division of the accumulator by the source operand.
  - uses the same registers as the DIV instruction

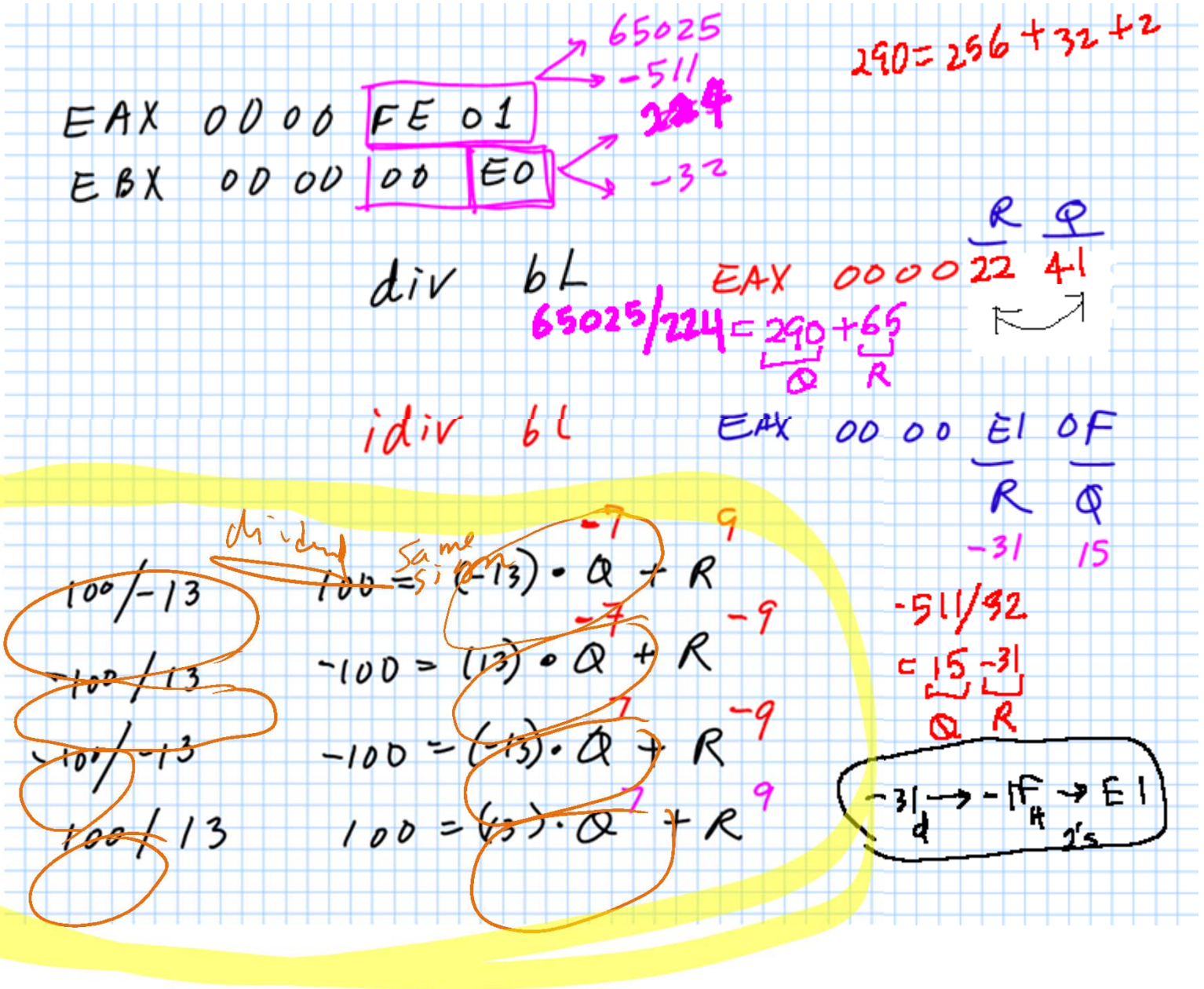
DIV opr/store summary



DIV & IDIV


EDX 00 00 00 00
EAX 00 00 00 64
EBX 00 00 00 0D
100/13
div ebx
EDX 00 00 00 09 R
EAX 00 00 00 07 Q
EAX 00 00 00 64
divisor (byte) 0D
div divisor
EAX 00 00 09 07
R Q
EDX 00 00 00 00
EAX 00 00 00 64
ECX FF FF FF F3
100/-13
idiv ecx
EDX 00 00 00 09 R
EAX FF FF FF F9 Q
EDX FF FF FF FF
EAX FF FF FF 9C
ECX 00 00 00 0D
-100/13
idiv ecx
EDX FF FF FF F7 R
EAX FF FF FF F9 Q

DIV vs IDIV


65025
-511
EAX 0000 FE01
EBX 0000 00E0
-32
290 = 256 + 32 + 2
div bL
65025/224 = 290 + 65
Q R
EAX 0000 22 41
R Q
idiv bL
EAX 0000 E1 0F
R Q
-31 15
dividend
same sign
100 = (-13) · Q + R
-100 = (13) · Q + R
-100 = (-13) · Q + R
100 = (13) · Q + R
100/-13
-100/13
-100/-13
100/13
-511/32
= 15 -31
Q R
-31 → -1F → E1
d H 2's

MUL & IMUL

DIV & IDIV

Exercise

A. MUL & IMUL Practice

(1) Before: [EAX]=FF FF FF E4; [EBX]=00 00 00 02
Instruction: MUL EBX
After: [EAX]=_____ [EDX]=_____

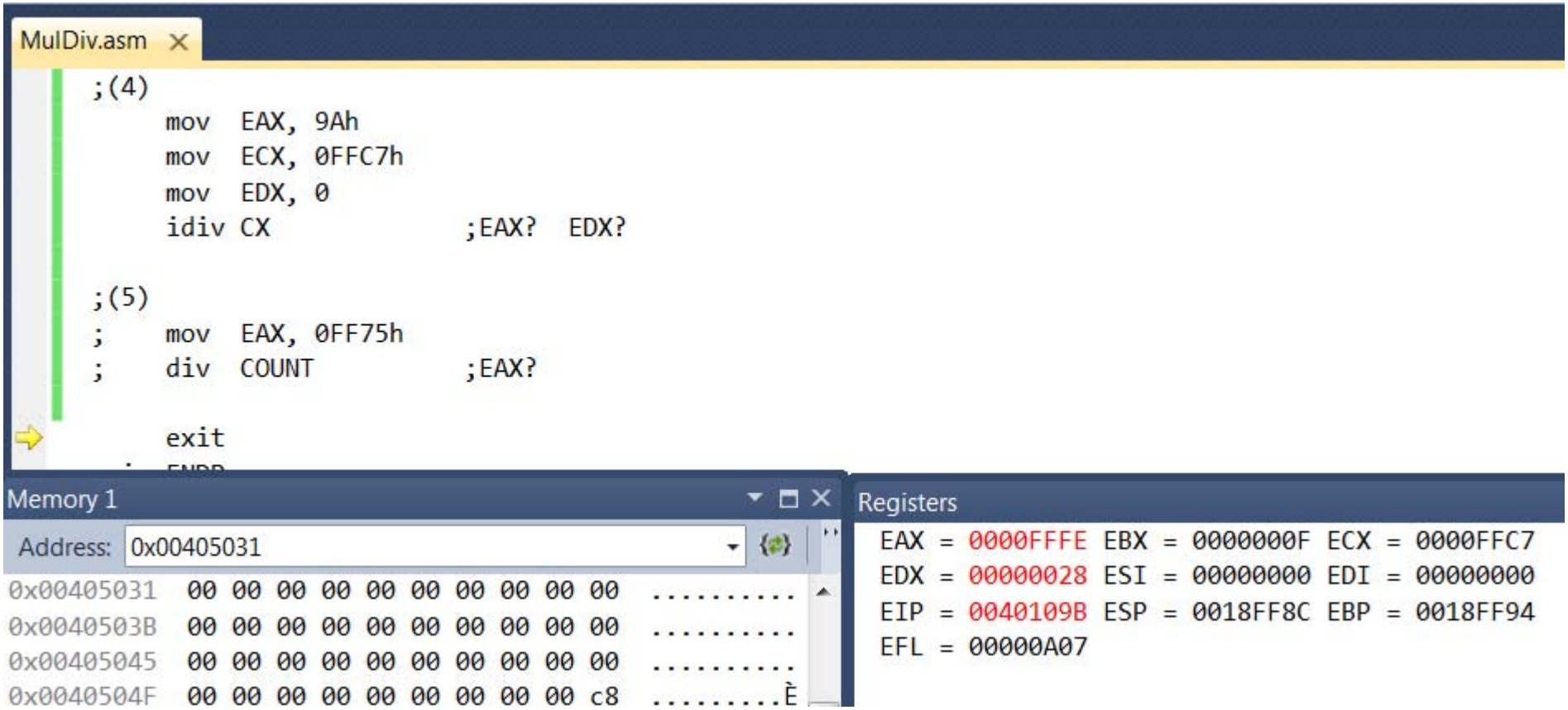
(2) Before: [EAX]=00 00 FF FF; [EDX]=FF FF 00 02
Instruction: MUL AX
After: [EAX]=_____ [EDX]=_____

B. DIV & IDIV Practice

(1) BEFORE: [EDX]=00 00 00 00, [EAX]=00 00 00 9A, [EBX]=00 00 00 0F
Instruction: IDIV EBX
AFTER: [EAX]=_____ [EDX]=_____

(2) BEFORE: [EAX]=00 00 FF 75, [COUNT]=FC {byte size}
Instruction: IDIV COUNT
AFTER: [EAX]=_____

MulDiv.asm


MulDiv.asm
;(4)
mov EAX, 9Ah
mov ECX, 0FFC7h
mov EDX, 0
idiv CX ;EAX? EDX?
;(5)
; mov EAX, 0FF75h
; div COUNT ;EAX?
exit
Memory 1
Address: 0x00405031
0x00405031 00 00 00 00 00 00 00 00 00 00 ....
0x0040503B 00 00 00 00 00 00 00 00 00 00 ....
0x00405045 00 00 00 00 00 00 00 00 00 00 ....
0x0040504F 00 00 00 00 00 00 00 00 00 c8 ....È
Registers
EAX = 0000FFFE EBX = 0000000F ECX = 0000FFC7
EDX = 00000028 ESI = 00000000 EDI = 00000000
EIP = 0040109B ESP = 0018FF8C EBP = 0018FF94
EFL = 00000A07

TcTf.asm

```
      mov  EDX,OFFSET prompt
      call WriteString

      call ReadDec
;Others -- ReadHex (Hex number), ReadInt (signed number)
      mov  TC,EAX

      mov  EAX, TC
      imul EAX, 9
      add  EAX,2
      mov  EBX,5
      cdq
      idiv EBX
      add  EAX,32
      mov  TF,eax

      exit
main ENDP
END main
```

Integer

$\frac{5}{2} \Rightarrow 2$

Round off → 3

$\frac{5+1}{2} \Rightarrow 3$

$\frac{1}{2}$ of divisor

100 %

Memory 1

Address: 0x00405000

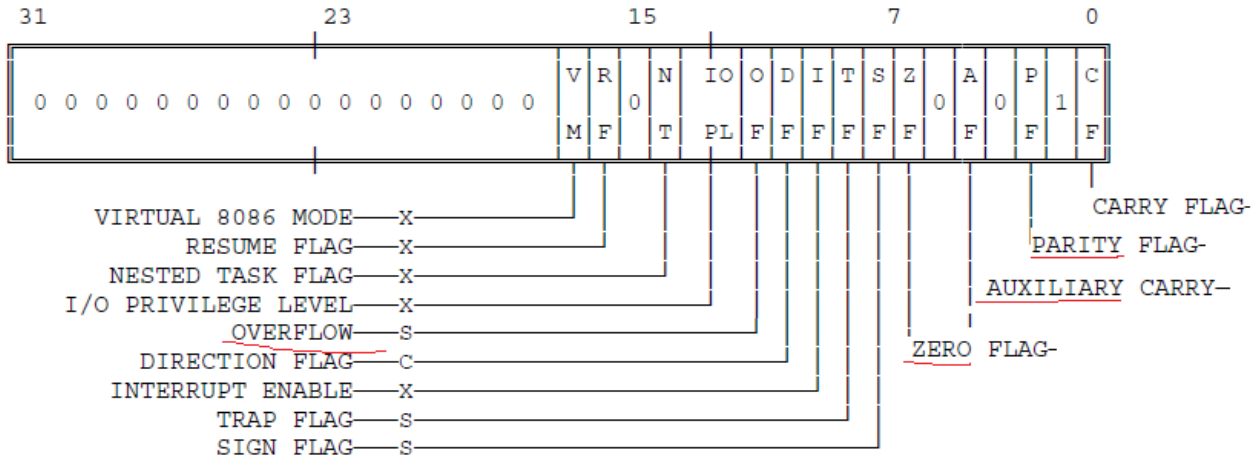
```
0x00405000  45 6e 74 65 72 20 79 6f  Enter yo
0x00405008  75 72 20 74 65 6d 70 65  ur tempe
0x00405010  72 61 74 75 72 65 20 69  rature i
0x00405018  6e 20 43 65 6c 63 69 75  n Celciu
0x00405020  73 20 28 54 63 29 3a 20  s (Tc):
0x00405028  20 00 20 00 00 00 5a 00   . ...Z.
```

Registers

```
EAX = 0000005A EBX = 00000005
ECX = 00000000 EDX = 00000000
ESI = 00000000 EDI = 00000000
EIP = 00401049 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000206
```

Boolean Operation Instruction

- AND, OR, XOR, and NOT
- NOT (Not)
  - inverts the bits in the specified operand to form a one's complement of the operand.
  - a unary operation that uses a single operand in a register or memory.
  - has no effect on the flags.
- AND: logical operation of "and“
- OR: Logical operation of “(inclusive)or”
- XOR: Logical operation of "exclusive or".
- AND, OR, XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.



Bit Test, Modify, Scan Instructions

- Bit Test
  - Operates on a single bit in a register or memory
  - assign the value of the selected bit to CF, the carry flag. Then a new value is assigned to the selected bit, as determined by the operation.

Instruction	Effect on CF	Effect on Selected Bit
Bit (Bit Test)	CF ← BIT	(none)
BTS (Bit Test and Set)	CF ← BIT	BIT ← 1
BTR (Bit Test and Reset)	CF ← BIT	BIT ← 0
BTC (Bit Test and Complement)	CF ← BIT	BIT ← NOT(BIT)

- Bit Scan
  - scan a word or doubleword for a one-bit and store the index of the first set bit into a register.
  - The ZF flag is set if the entire word is zero (no set bits are found)
  - ZF is cleared if a one-bit is found.
  - If no set bit is found, the value of the destination register is undefined.
  - BSF (Bit Scan Forward)
    - scans from low-order to high-order (starting from bit index zero).
  - BSR (Bit Scan Reverse)
    - scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).

Shift Instructions

- The bits in bytes, words, and doublewords may be shifted arithmetically or logically, up to 31 places.
- Specification of the count of shift
  - Implicitly as a single shift
  - Immediate value
  - Value contained in the CL (lower order 5 bits)
- **CF** always contains the value of the last bit shifted out of the destination operand.
- In a single-bit shift, **OF** is set if the value of the high-order (**sign**) bit was changed by the operation. Otherwise, OF is cleared.
- The shift instructions provide a convenient way to **accomplish division or multiplication by binary power**.

SAL, SAR, SHL, SHR

- SAL (Shift Arithmetic Left) shifts the destination byte, word, or doubleword operand left by one or by the number of bits specified in the count operand.
  - CF receives the last bit shifted out of the left of the operand.
- SAR (Shift Arithmetic Right) shifts the destination byte, word, or doubleword operand to the right by one or by the number of bits specified in the count operand.
  - SAR **preserves the sign** of the register/mem operand as it shifts the operand to the right.
  - CF receives the last bit shifted out of the right of the operand.
- SHL (Shift Logical Left) is a synonym for SAL
  - CF Receives the last bit shifted out of the left of the operand.
  - SHL shifts in zeros to fill the vacated bit locations
- SHR (Shift Logical Right) shifts the destination byte, word, or doubleword operand right by one or by the number of bits specified in the count operand.
  - CF received the last bit shifted out of the right of the operand.
  - Shifts in zeros to fill the vacated bit locations.

SHL SAL SHR SAR

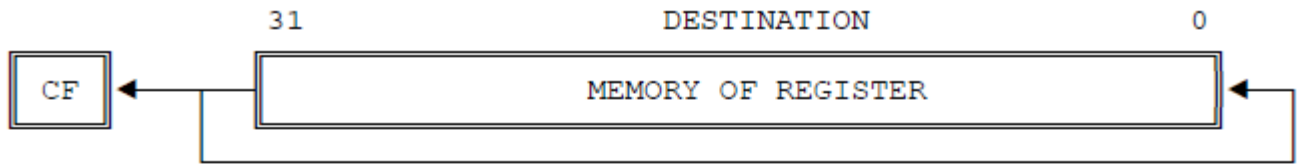
	OF	CF	OPERAND
BEFORE SHL OR SAL	X	X	10001000100010001000100010001111
AFTER SHL OR SAL BY 1	1	1 ←	00010001000100010001000100011110 ← 0
AFTER SHL OR SAL BY 10	X	0 ←	00100010001000100011110000000000 ← 0

		OPERAND	CF
BEFORE SHR		10001000100010001000100010001111	X
AFTER SHR BY 1	0→	01000100010001000100010001000111	→1
AFTER SHR BY 10	0→	00000000001000100010001000100010	→0

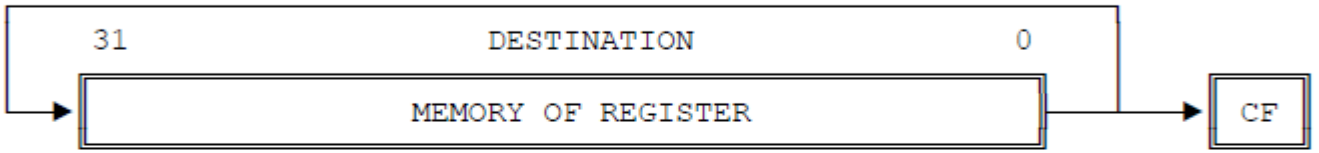
		POSITIVE OPERAND	CF
BEFORE SAR		01000100010001000100010001000111	X
AFTER SAR BY 1	0→	00100010001000100010001000100011	→1
		NEGATIVE OPERAND	CF
BEFORE SAR		11000100010001000100010001000111	X
AFTER SAR BY 1	0→	11100010001000100010001000100011	→1

Rotation

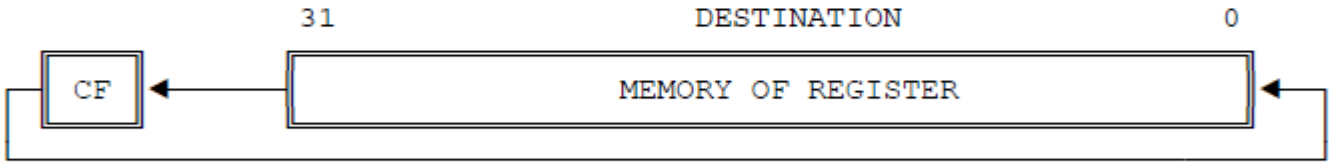
ROL



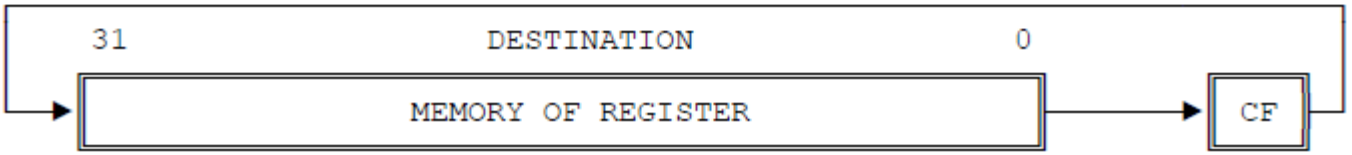
ROR



RCL



RCR

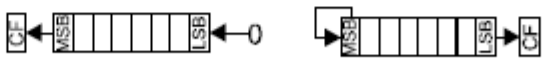
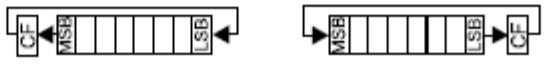
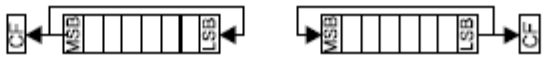


MOVE

TRANSFER				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:= ¬CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±				±	±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN *i*	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port									
OUT *i*	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

i for more information see instruction specifications Flags: ±=affected by this instruction ?=undefined after this instruction

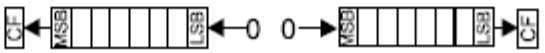
Arithmetic

ARITHMETIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	
CMP	Compare	CMP Op1,Op2	Op1-Op2	±				±	±	±	±	±
SAL	Shift arithmetic left (≡ SHL)	SAL Op,Quantity		*i*				±	±	?	±	±
SAR	Shift arithmetic right	SAR Op,Quantity		*i*				±	±	?	±	±
RCL	Rotate left through Carry	RCL Op,Quantity		*i*								±
RCR	Rotate right through Carry	RCR Op,Quantity		*i*								±
ROL	Rotate left	ROL Op,Quantity		*i*								±
ROR	Rotate right	ROR Op,Quantity		*i*								±

i for more information see instruction specifications

♦ then CF:=0, OF:=0 else CF:=1, OF:=1

Logic +

LOGIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=¬Op (invert each bit)									
AND	Logical and	AND Dest,Source	Dest:=Dest∧Source	0				±	±	?	±	0
OR	Logical or	OR Dest,Source	Dest:=Dest∨Source	0				±	±	?	±	0
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (exor) Source	0				±	±	?	±	0
SHL	Shift logical left (≡ SAL)	SHL Op,Quantity		*i*				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity		*i*				±	±	?	±	±

MISC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NOP	No operation	NOP	No operation									
LEA	Load effective address	LEA Dest,Source	Dest := address of Source									
INT	Interrupt	INT Nr	interrupts current program, runs spec. int-program			0	0					

Jump

JUMPS (flags remain unchanged)			
Name	**Comment**	**Code**	**Operation**
CALL	Call subroutine	CALL Proc	
JMP	Jump	JMP Dest	
JE	Jump if Equal	JE Dest	(≡ JZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)
JCXZ	Jump if CX Zero	JCXZ Dest	
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)

Name	**Comment**	**Code**	**Operation**
RET	Return from subroutine	RET	
JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JECXZ	Jump if ECX Zero	JECXZ Dest	
JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

JUMPS Unsigned (Cardinal)			
JA	Jump if Above	JA Dest	(≡ JNBE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)
JNA	Jump if not Above	JNA Dest	(≡ JBE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)
JC	Jump if Carry	JC Dest	
JNC	Jump if no Carry	JNC Dest	

JUMPS Signed (Integer)			
JG	Jump if Greater	JG Dest	(≡ JNLE)
JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JL	Jump if Less	JL Dest	(≡ JNGE)
JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JO	Jump if Overflow	JO Dest	
JNO	Jump if no Overflow	JNO Dest	
JS	Jump if Sign (= negative)	JS Dest	
JNS	Jump if no Sign (= positive)	JNS Dest	

General Registers: