x86 Assembly Programming

EECE416 uC

Resources:

Intel 80386 Programmers Reference Manual

Essentials of 80x86 Assembly Language

Introduction to 80x86 Assembly Language Programming

Registers for x86

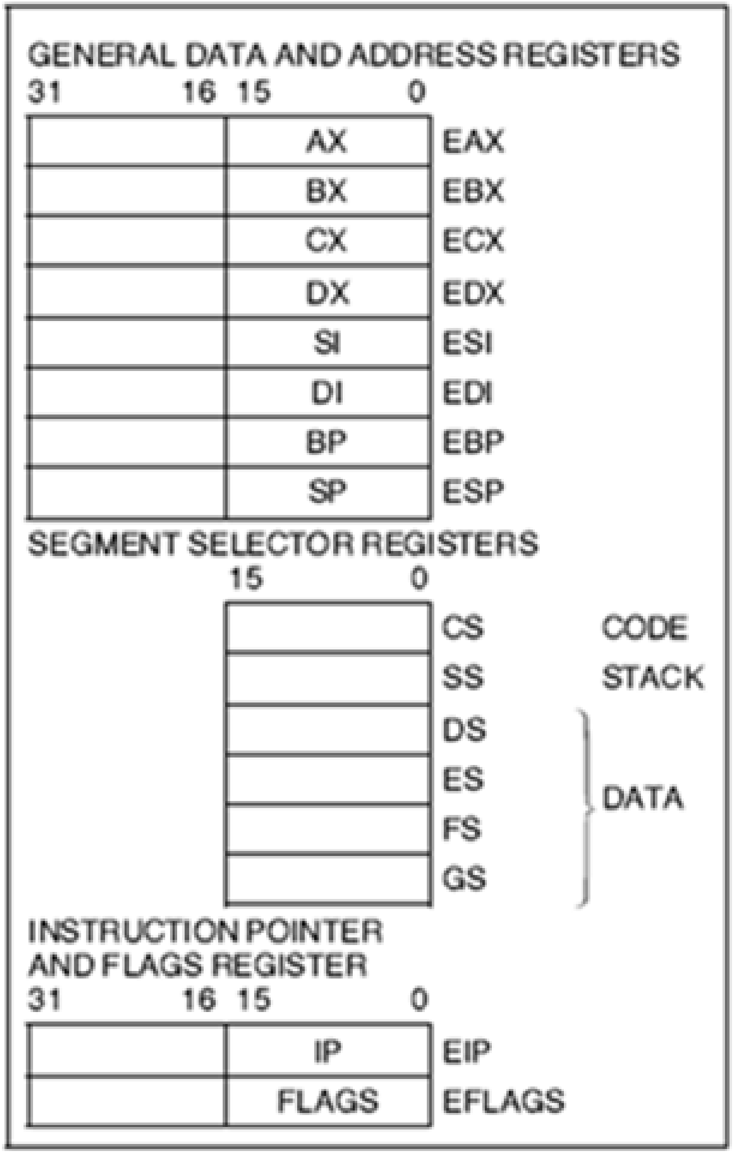
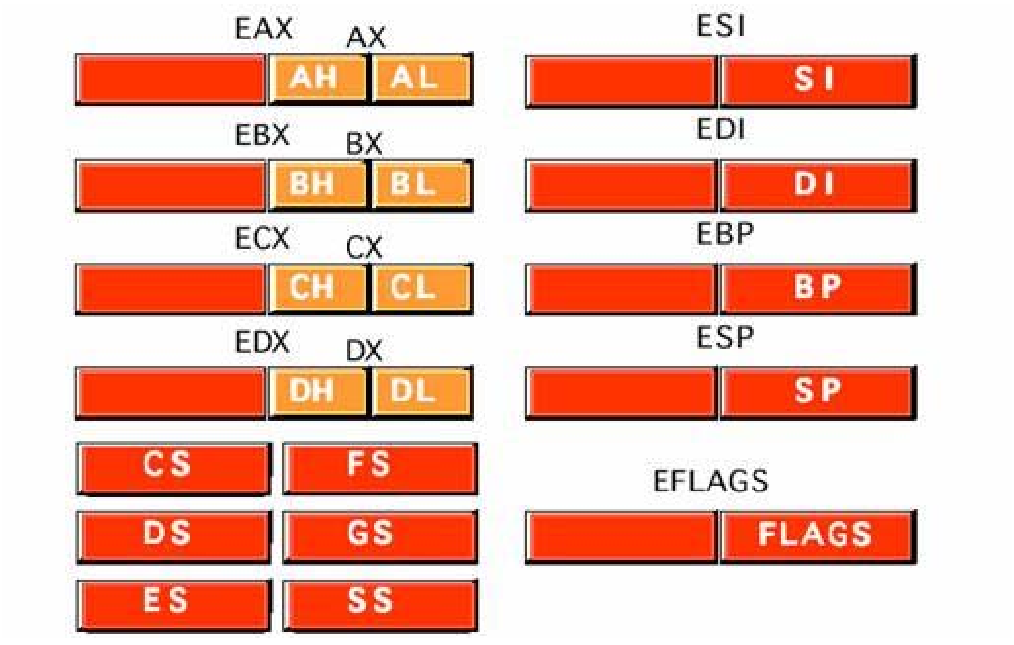
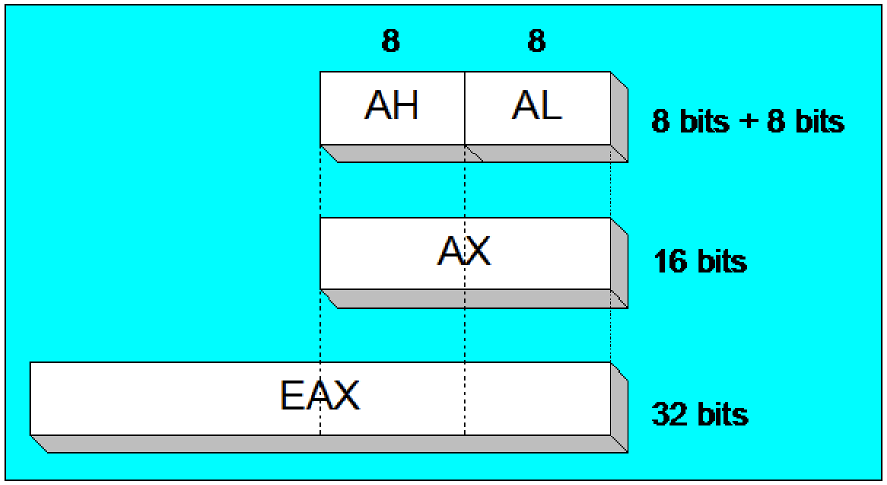


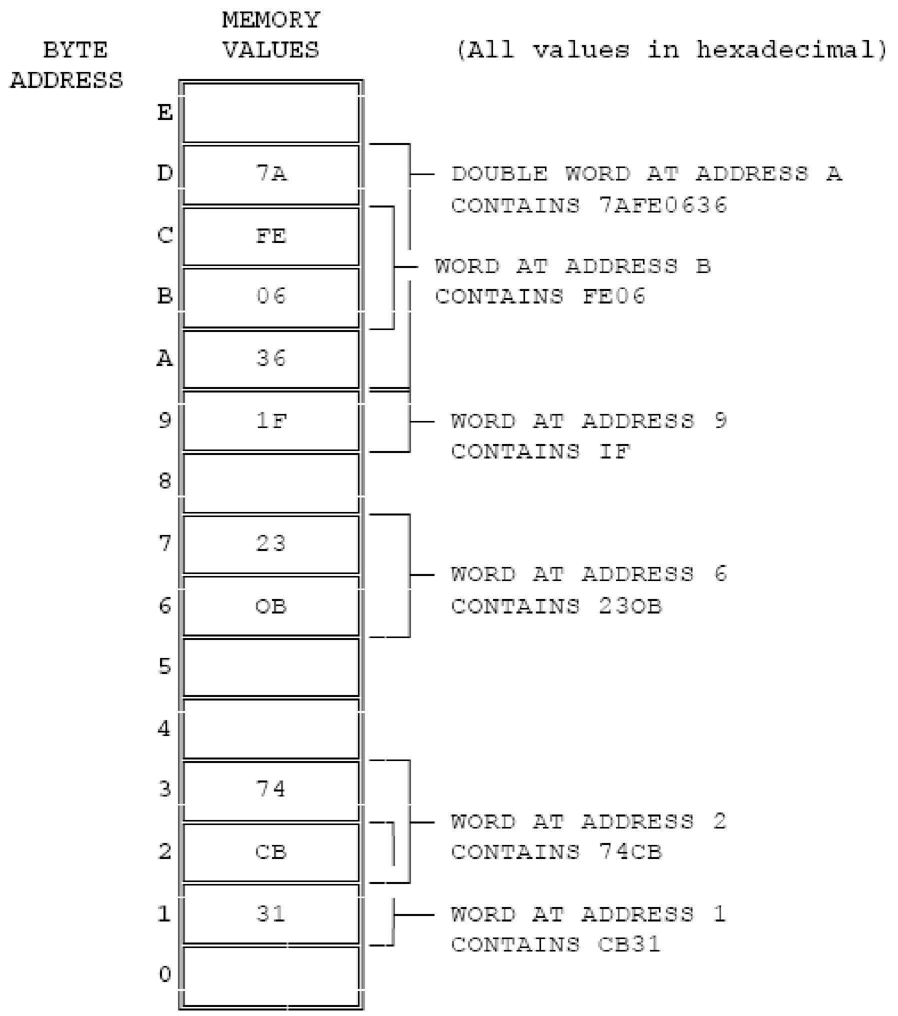
Figure 2-1. Intel386™ DX Base Architecture Registers

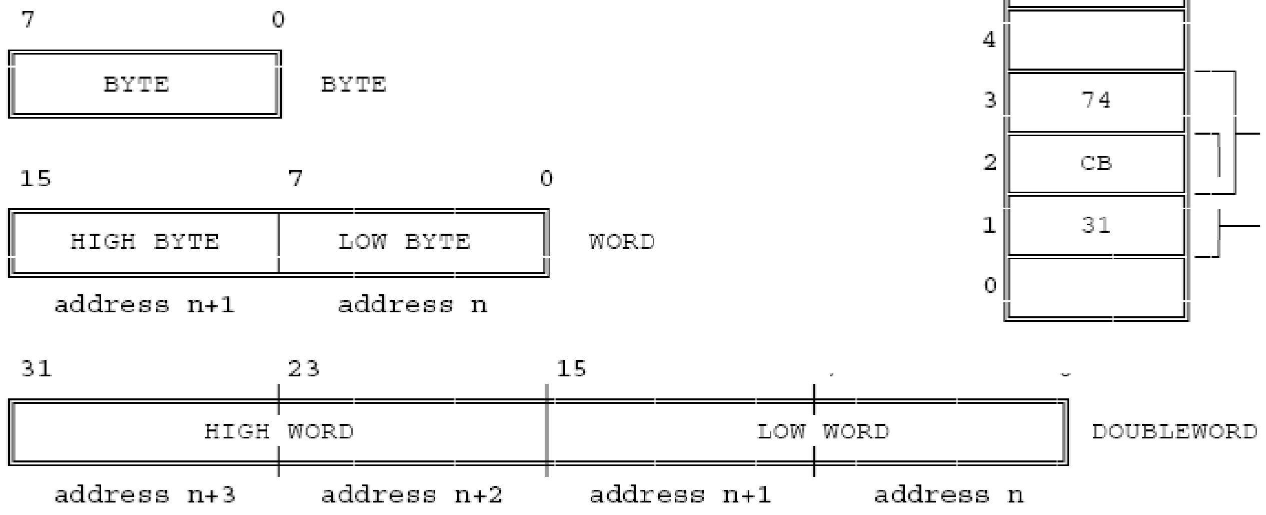




Basic Data Types

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

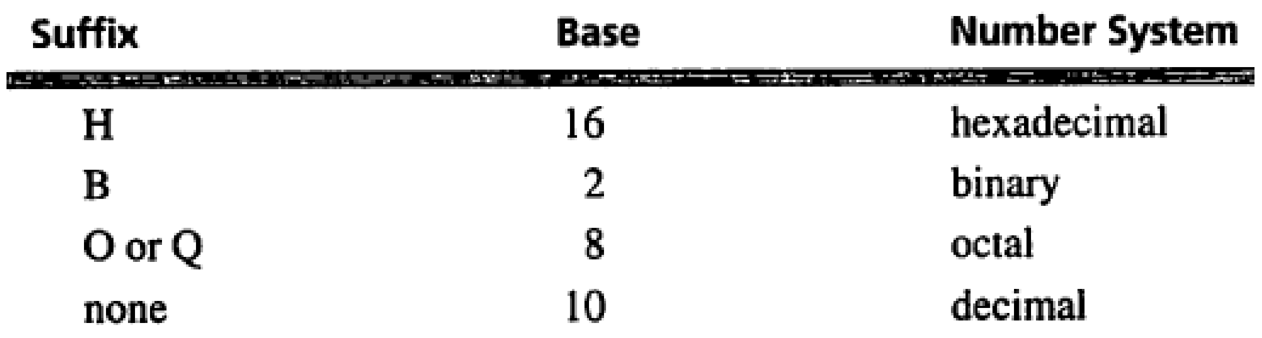

BYTE ADDRESS
MEMORY VALUES
(All values in hexadecimal)
E
D 7A
C FE
B 06
A 36
9 1F
8
7 23
6 OB
5
4
3 74
2 CB
1 31
0
DOUBLE WORD AT ADDRESS A CONTAINS 7AFE0636
WORD AT ADDRESS B CONTAINS FE06
WORD AT ADDRESS 9 CONTAINS IF
WORD AT ADDRESS 6 CONTAINS 23OB
WORD AT ADDRESS 2 CONTAINS 74CB
WORD AT ADDRESS 1 CONTAINS CB31



Other Data Types

- Integer
  - Signed numeric: 8, 16, or 32 bits
  - 2’s complement representation
  - MSb: sign bit
- Ordinal
  - Unsigned numeric: 8, 16 or 32 bits
- Near Pointer
  - 32-bit logical address
  - Offset within a segment
- Far Pointer
  - 48-bit address space with segment selector + offset
- String
  - 8, 16, or 32 bits

Data Declaration



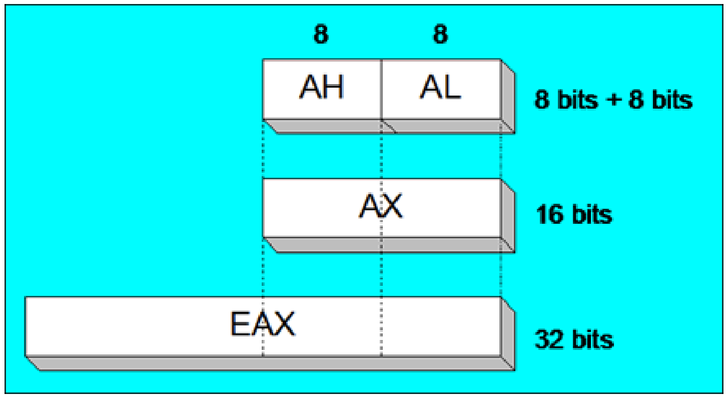
Suffix	Base	Number System
H	16	hexadecimal
B	2	binary
O or Q	8	octal
none	10	decimal

- 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     -1
      num2    WORD     FFFFH
      ```
  - DWORD: 4 bytes are reserved
    - Example:
      ```
      N1       DWORD    -1
      ```
  - QWORD: 8 bytes
    - 64 bit: RAX RBX RCX ,etc
    - 32 bit: **EDX:EAX** Concatenation for **CDQ** instruction

Register Size and Data

- Assuming that the content of **eax** is [01FF01FF], what would be the content of **eax** after each instruction?

```
mov     al, 155
mov     ax, 155
mov     eax, 155
```



- label   mnemonic   dst, src

Exercise of Register Size

- Handout

Instruction Format

- Opcode:
  - specifies the operation performed by the instruction.
- Register specifier
  - an instruction may specify one or two register operands.

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

- 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.

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

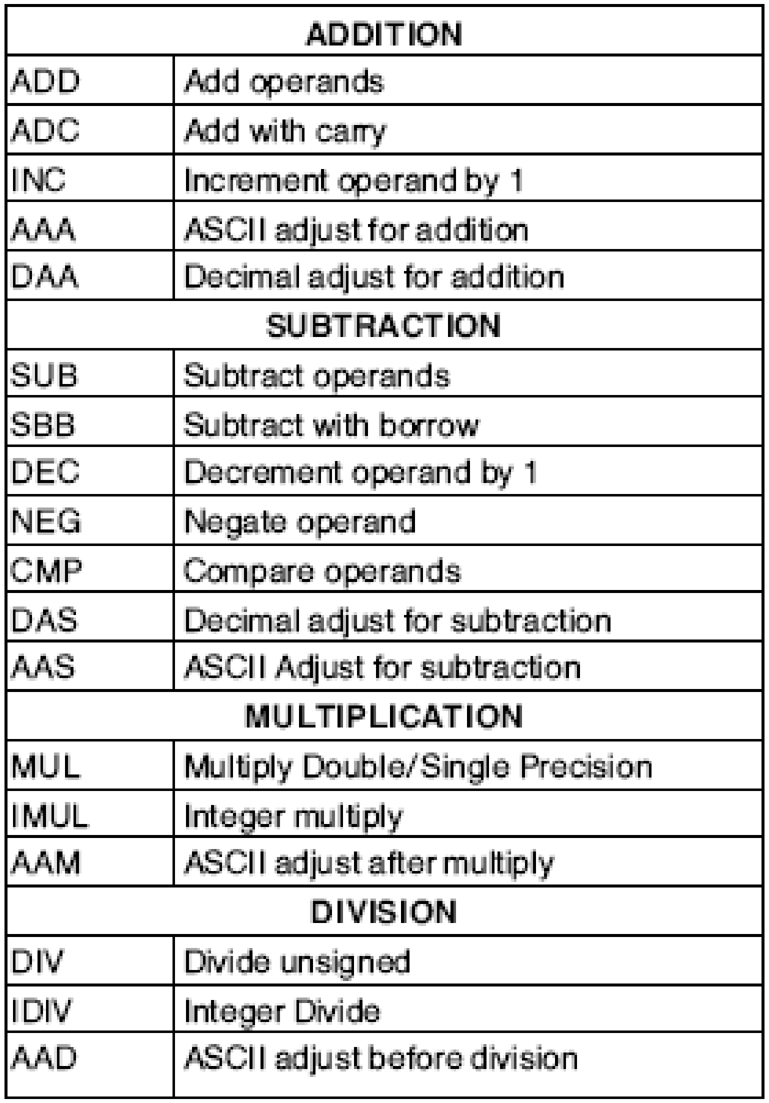
- Immediate operand
  - when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide.

```
mov   eax, 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

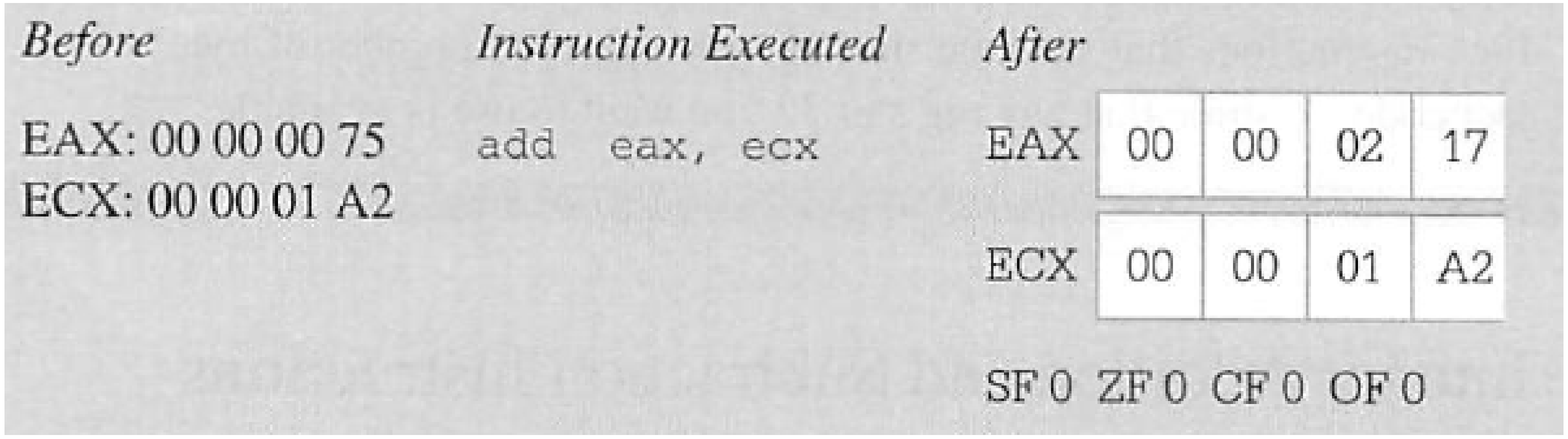
- MOV (Move)
  - transfers a byte, word, or doubleword from the source operand to the destination operand: R→ M. M → R, R→ R, I→R, I→ M
  - The MOV instruction cannot move M→M or from SR → SR (segment register)
  - M→M via MOVS (string)
- XCHG (Exchange)
  - swaps the contents of two operands.
  - swap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

Type Conversion Instruction

- 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.
- 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**.
- MOVSX (Move with Sign Extension)
  - sign-extends an 8-bit value to a 16-bit value and a 8- or 16-bit value to 32-bit value.
- MOVZX (Move with Zero Extension)
  - extends an 8-bit value to a 16-bit value and an 8- or 16-bit value to 32-bit value by inserting high-order zeros.

Addition Instruction

- ADD (Add Integers)
  - replaces the destination operand with the sum of the source and destination operands. Sets CF if overflow.
- ADC (Add Integers with Carry)
  - sums the operands, adds one if CF is set, and replaces the destination operand with the result. If CF is cleared, ADC performs the same operation as the ADD instruction. An ADD followed by multiple ADC instructions can be used to add numbers longer than 32 bits.



Before

EAX: 00 00 00 75
ECX: 00 00 01 A2

Instruction Executed

```
add   eax, ecx
```

After

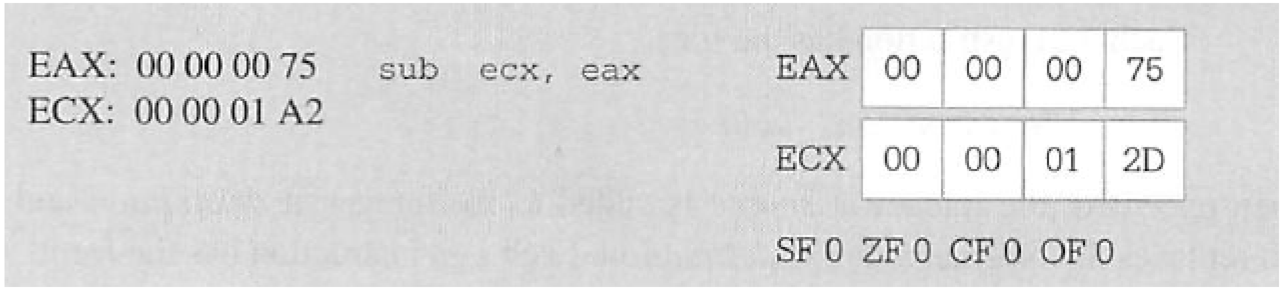
EAX	00	00	02	17
ECX	00	00	01	A2

SF 0 ZF 0 CF 0 OF 0

- label mnemonic dst, src

SUB (Subtract Integers)

- SUB:
  - 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.
- SBB (Subtract Integers with Borrow)
  - subtracts the source operand from the destination operand, subtracts 1 if CF is set, and returns the result to the destination operand. If CF is cleared, SBB performs the same operation as SUB. SUB followed by multiple SBB instructions may be used to subtract numbers longer than 32 bits. If CF is cleared, SBB performs the same operation as SUB.



```
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

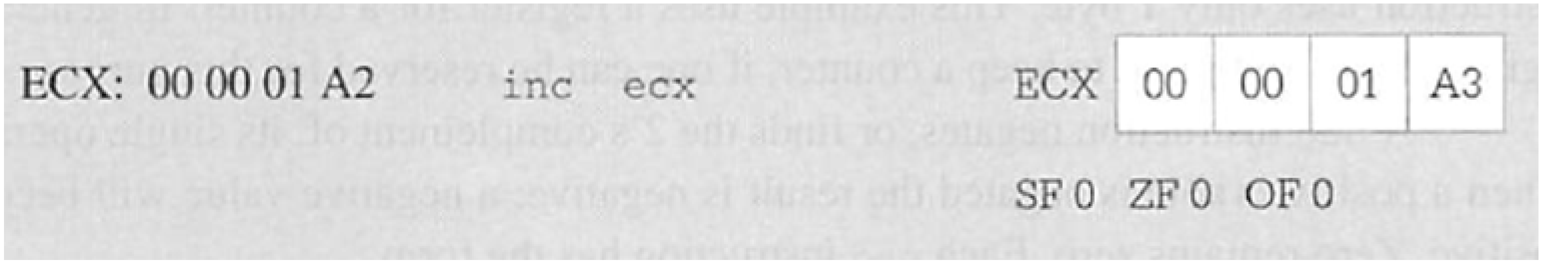
- label   mnemonic   dst, src

ADD & SUB Examples

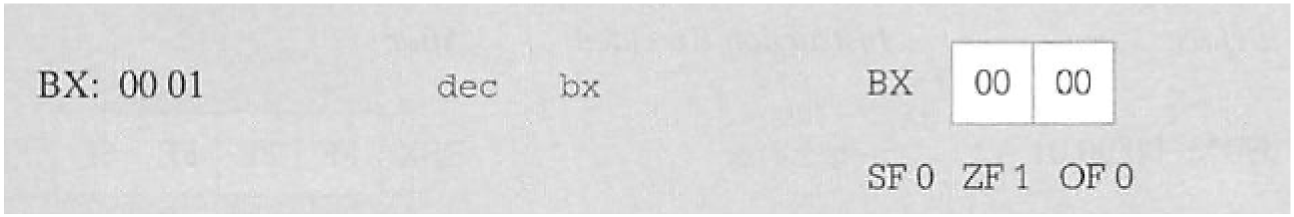
EAX: 00 00 00 75 ECX: 00 00 01 A2	`sub eax, ecx`	EAX | FF | FF | FE | D3 ECX | 00 | 00 | 01 | A2	SF 1 ZF 0 CF 0 OF 0
AX: 77 AC CX: 4B 35	`add ax, cx`	AX | C2 | E1 CX | 4B | 35	SF 1 ZF 0 CF 0 OF 1
EAX: 00 00 00 75 ECX: 00 00 01 A2	`sub ecx, eax`	EAX | 00 | 00 | 00 | 75 ECX | 00 | 00 | 01 | 2D	SF 0 ZF 0 CF 0 OF 0
BL: 4B	`add bl, 4`	BL | 4F	SF 0 ZF 0 CF 0 OF 0
DX: FF 20 word at value: FF 20	`sub dx, Value`	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: `00 00 01 00`	`sub Dbl, 1`	Dbl | 00 | 00 | 00 | FF	SF 0 ZF 0 CF 0 OF 0

INC & DEC

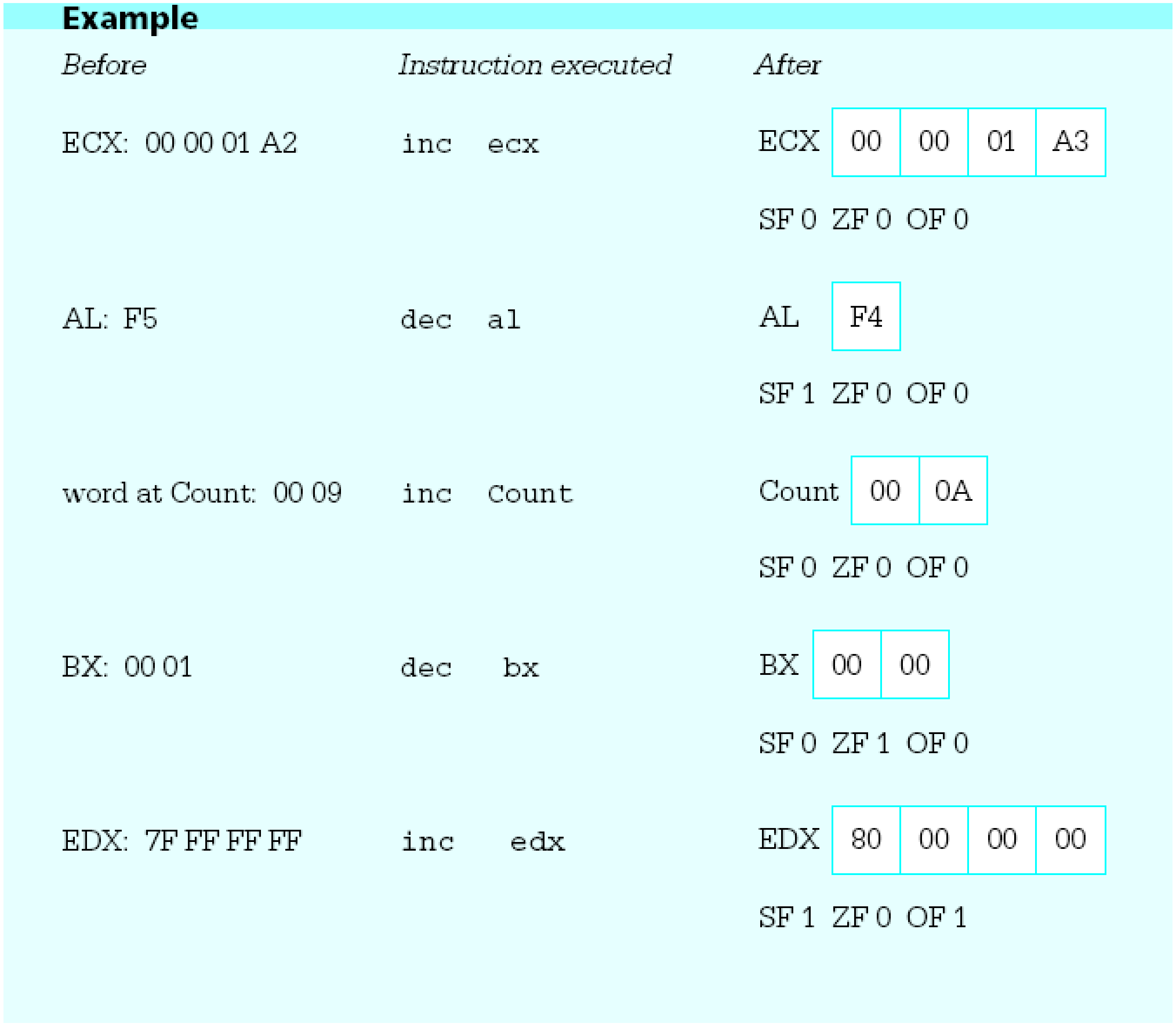
- INC (Increment)
  - adds one to the destination operand. INC does not affect CF. Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed.



- DEC (Decrement)
  - 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

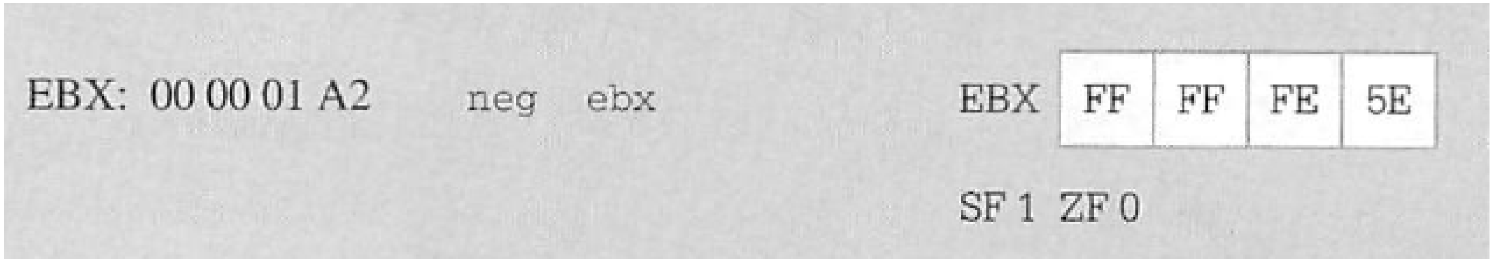
Before	*Instruction executed*	*After*
ECX: 00 00 01 A2	`inc ecx`	ECX 00 00 01 A3 SF 0 ZF 0 OF 0
AL: F5	`dec al`	AL F4 SF 1 ZF 0 OF 0
word at Count: 00 09	`inc Count`	Count 00 0A SF 0 ZF 0 OF 0
BX: 00 01	`dec bx`	BX 00 00 SF 0 ZF 1 OF 0
EDX: 7F FF FF FF	`inc edx`	EDX 80 00 00 00 SF 1 ZF 0 OF 1

CMP + NEG

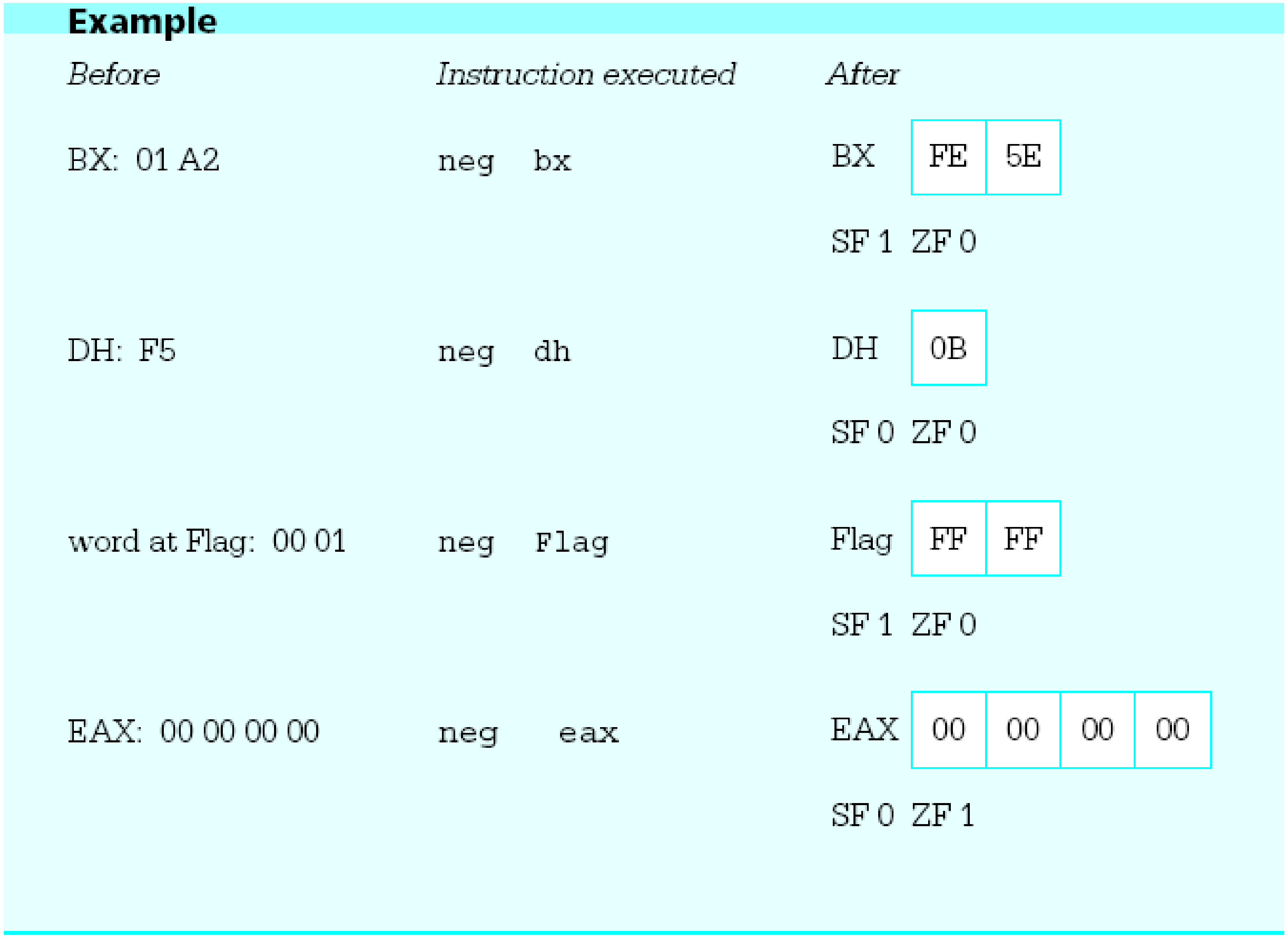
- CMP (Compare)
  - **subtracts** the source operand from the destination operand. It updates OF, SF, ZF, AF, PF, and CF but does not alter the source and destination operands.

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

- NEG (Negate)
  - **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**.**



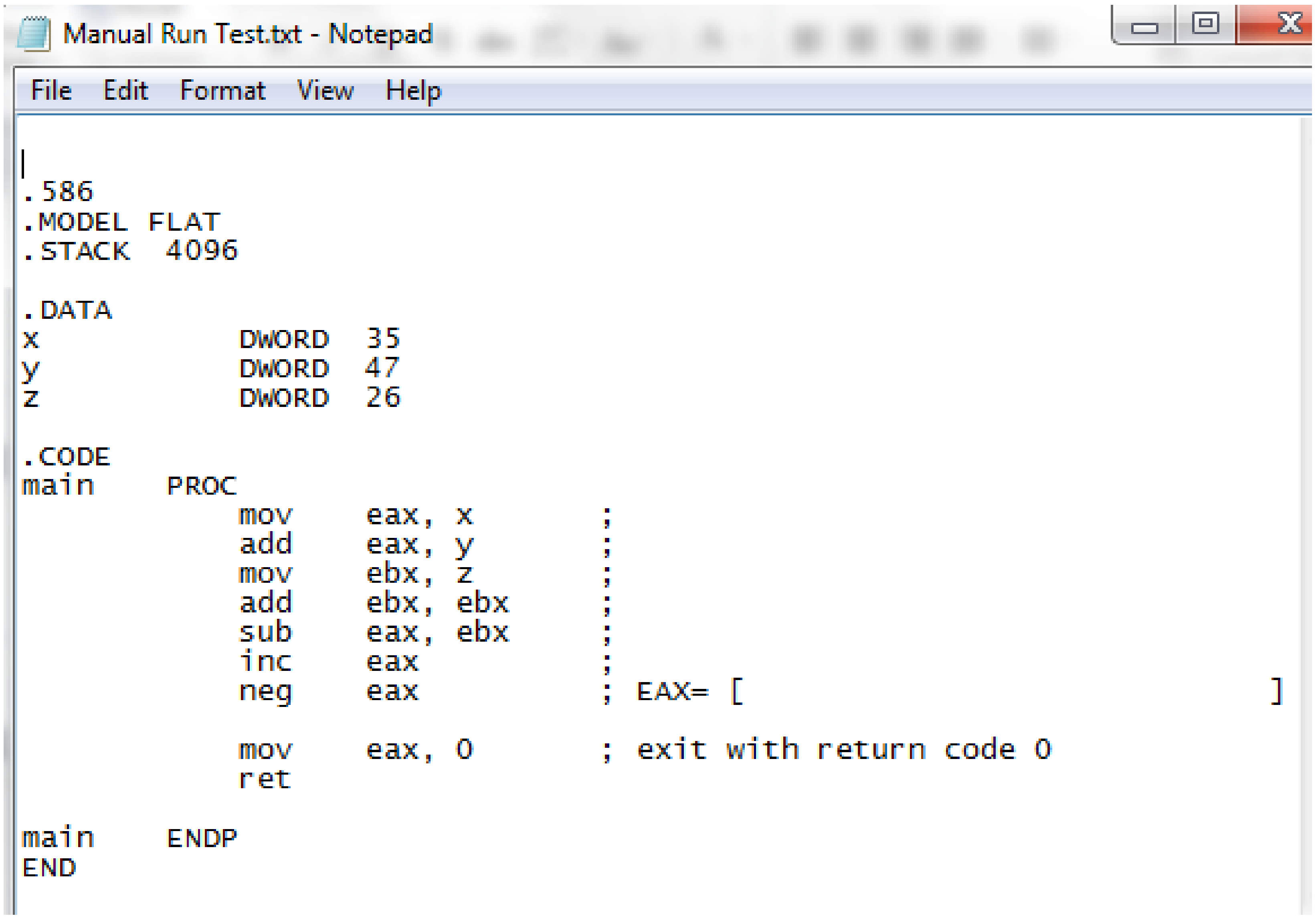
NEG Examples



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

Manual execution practice –Done !

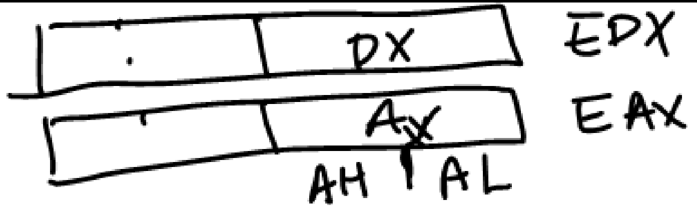


```
.586
.MODEL FLAT
.STACK  4096

.DATA
x               DWORD   35
y               DWORD   47
z               DWORD   26

.CODE
main      PROC
                mov     eax, x          ;
                add     eax, y          ;
                mov     ebx, z          ;
                add     ebx, ebx        ;
                sub     eax, ebx        ;
                inc     eax             ;
                neg     eax             ; EAX= [                              ]

                mov     eax, 0          ; exit with return code 0
                ret

main      ENDP
END
```

Multiplication Instruction - MUL

- 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.
  - **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. 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



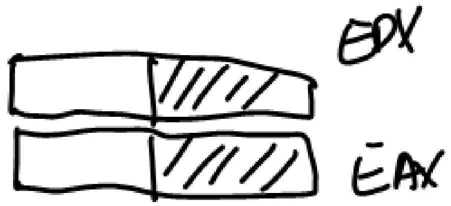
MUL byte

$\xrightarrow[\text{opr}]{} [AL] \times [byte] \xrightarrow[\text{store}]{} [AX]$

word

MUL BX

$\xrightarrow[\text{opr}]{} [AX] \times [BX] \xrightarrow[\text{store}]{}$

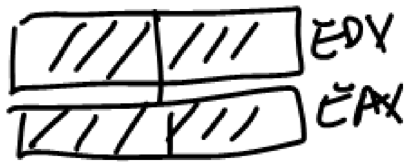


[DX | AX]

D-word

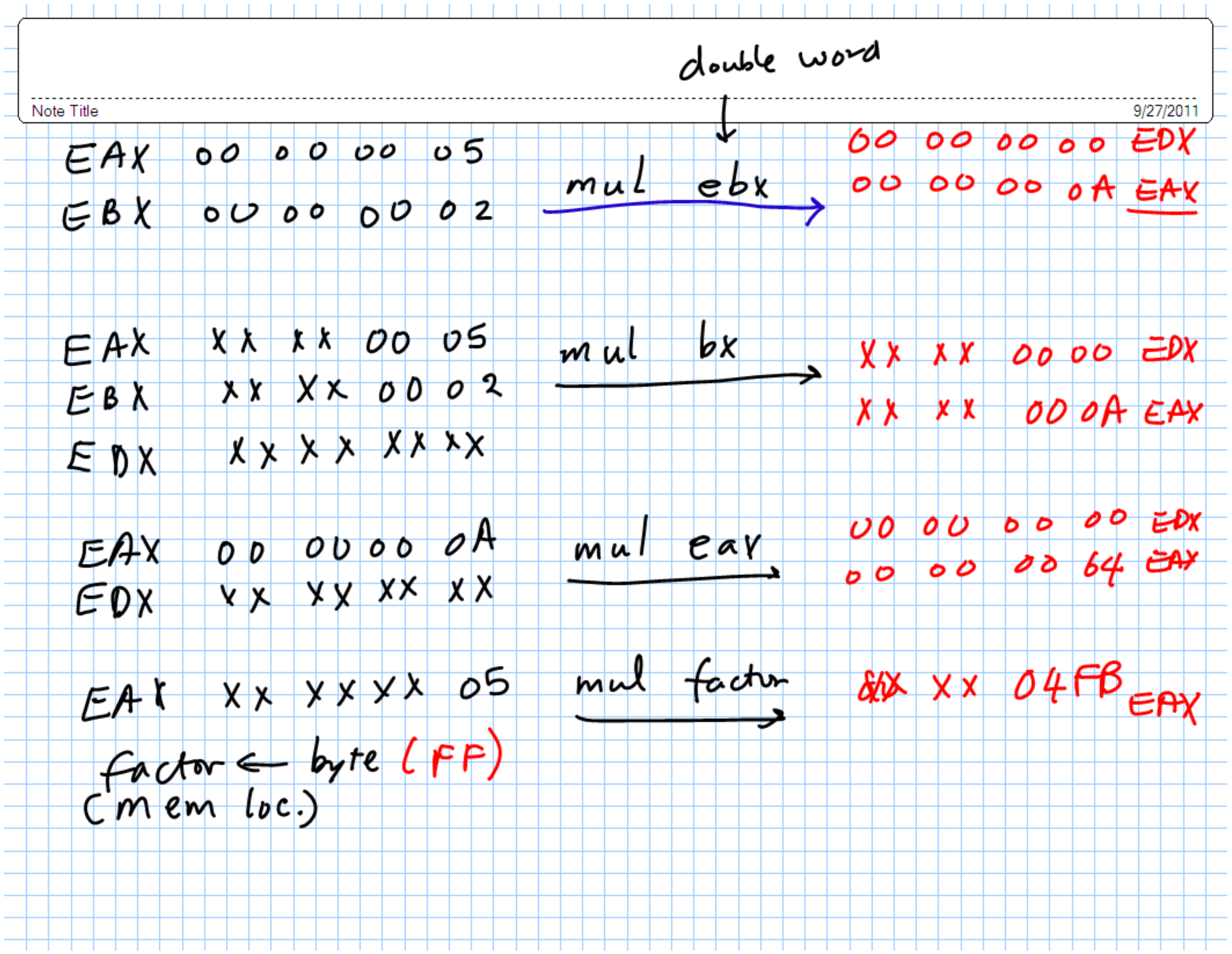
MUL EBX

$\xrightarrow[\text{opr}]{} [EAX] \times [EBX] \xrightarrow[\text{store}]{}$



[EDX | EAX]

MUL - Exercise



Note Title 9/27/2011

double word

EAX 00 00 00 05
EBX 00 00 00 02
$\xrightarrow{\text{mul ebx}}$
00 00 00 00 EDX
00 00 00 0A EAX

EAX XX XX 00 05
EBX XX XX 00 02
EDX XX XX XX XX
$\xrightarrow{\text{mul bx}}$
XX XX 00 00 EDX
XX XX 00 0A EAX

EAX 00 00 00 0A
EDX XX XX XX XX
$\xrightarrow{\text{mul eax}}$
00 00 00 00 EDX
00 00 00 64 EAX

EAX XX XX XX 05
$\xrightarrow{\text{mul factor}}$
XX XX 04FB EAX

factor ← byte (FF)
(mem loc.)

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.**

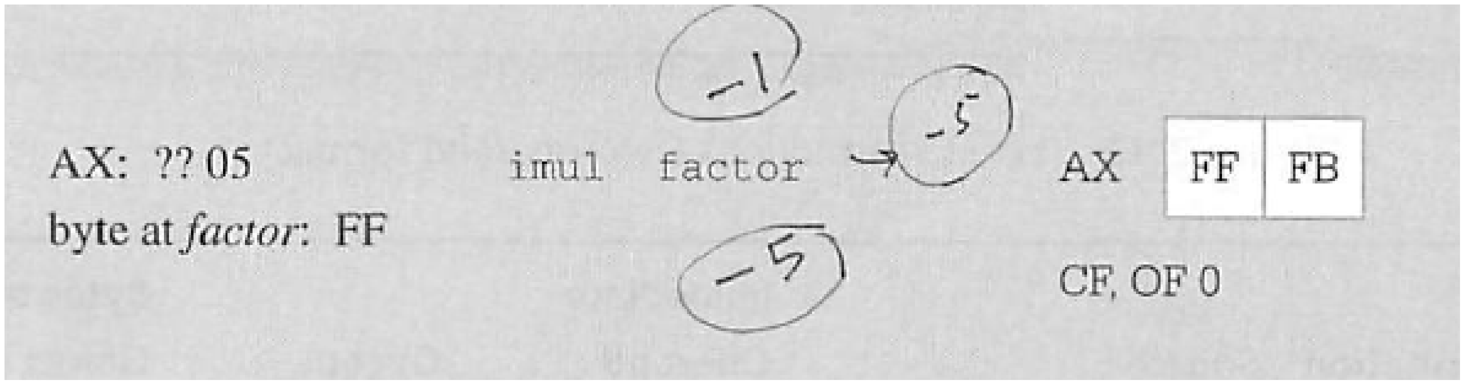
IMUL

Eax 00 00 00 05
Ebx 00 00 00 02
$\xrightarrow{\text{imul ebx}}$
Edx 00 00 00 00
eax 00 00 00 0A

Eax XX XX 00 05
Ebx XX XX 00 02
$\xrightarrow{\text{imul bx}}$
Edx xx xx 00 00
eax xx xx 00 0A

EAX XX XX XX 05
factor (FF)
$\xrightarrow{\text{imul factor}}$
EAX xx xx FF FB

Ebx 00 00 00 0A
$\xrightarrow{\text{imul ebx, 10}}$
ebx 00 00 00 64



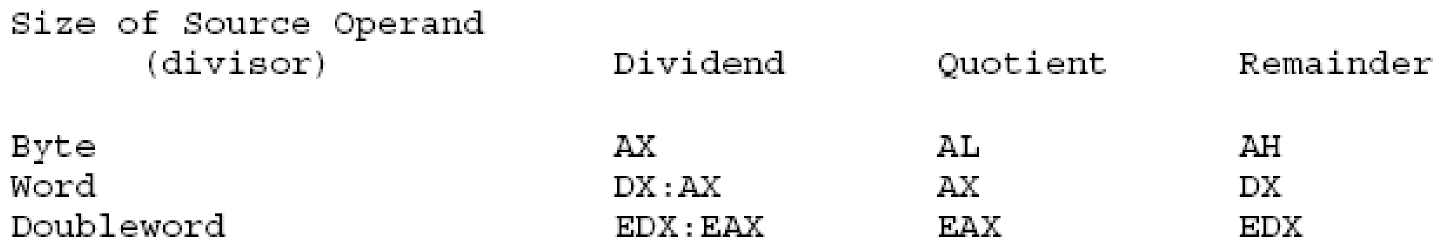
AX: ?? 05
byte at *factor*: FF
imul factor → AX FF FB
CF, OF 0

MUL & IMUL Exercise

MUL IMUL Practice F12.docx

Division Instruction

- 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)
  - the quotient and remainder have the same size as the divisor.



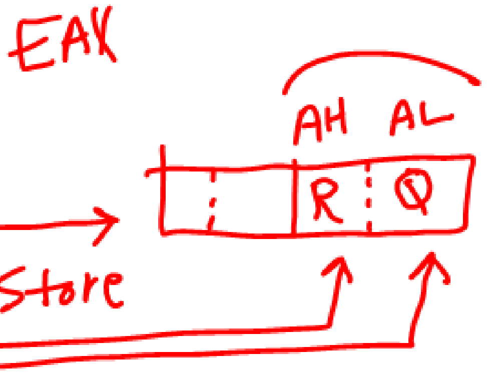
Size of Source Operand (divisor)	Dividend	Quotient	Remainder
Byte	AX	AL	AH
Word	DX:AX	AX	DX
Doubleword	EDX:EAX	EAX	EDX

- 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

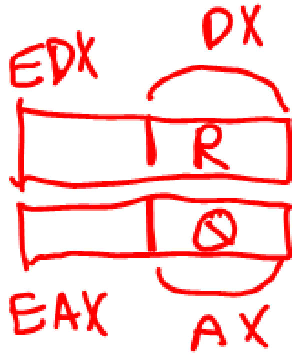
- Div byte

$$\xrightarrow[\text{opr}]{} \frac{[AX]}{[byte]} = Q + R \xrightarrow[\text{Store}]{}$$



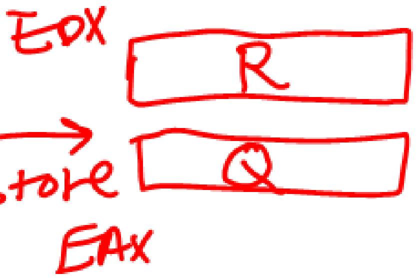
- Div Word

$$\xrightarrow[\text{opr}]{} \frac{[DX \mid AX]}{[WORD]} = Q + R \xrightarrow[\text{Store}]{}$$



- Div D-word

$$\xrightarrow[\text{opr}]{} \frac{[EDX \mid EAX]}{[D\text{-}WORD]} = Q + R \xrightarrow[\text{store}]{}$$



DIV & IDIV

```
EDX  00 00 00 00                    100/13
EAX  00 00 00 64    div ebx         EDX  00 00 00 09   R
EBX  00 00 00 0D   -------->        EAX  00 00 00 07   Q
```

```
EAX  00 00 00 64    div divisor     EAX  00 00 09 07
                   ---------->                 R  Q
divisor  0D
(byte)
```

```
                                     100/-13
EDX  00 00 00 00    idiv ecx        EDX  00 00 00 09   R
EAX  00 00 00 64   -------->        EAX  FF FF FF F9   Q
ECX  FF FF FF F3
```

```
                                 -100/13
EDX  FF FF FF FF    idiv ecx        EDX  FF FF FF F7   R
EAX  FF FF FF 9C   -------->        EAX  FF FF FF F9   Q
ECX  00 00 00 0D
```

DIV vs IDIV

EAX 0000 $\boxed{FE01}$ → 65025, −511

EBX 0000 $\boxed{00}\,\boxed{E0}$ → ~~224~~, −32

$290 = 256 + 32 + 2$

div bL

$65025/224 = \underbrace{290}_{Q} + \underbrace{65}_{R}$

EAX 0000 $\overset{R}{22}$ $\overset{Q}{41}$

idiv bl

EAX 0000 $\underset{R}{E1}$ $\underset{Q}{0F}$

-31 15

$-511/32 = \underbrace{15}_{Q}\ \underbrace{-31}_{R}$

$\boxed{-31_d \rightarrow -1F_H \rightarrow E1_{2's}}$

dividend, same sign

$100/-13$ $\quad 100 = (-13)\cdot Q + R$ $\quad Q = -7,\ R = 9$

$-100/13$ $\quad -100 = (13)\cdot Q + R$ $\quad Q = -7,\ R = -9$

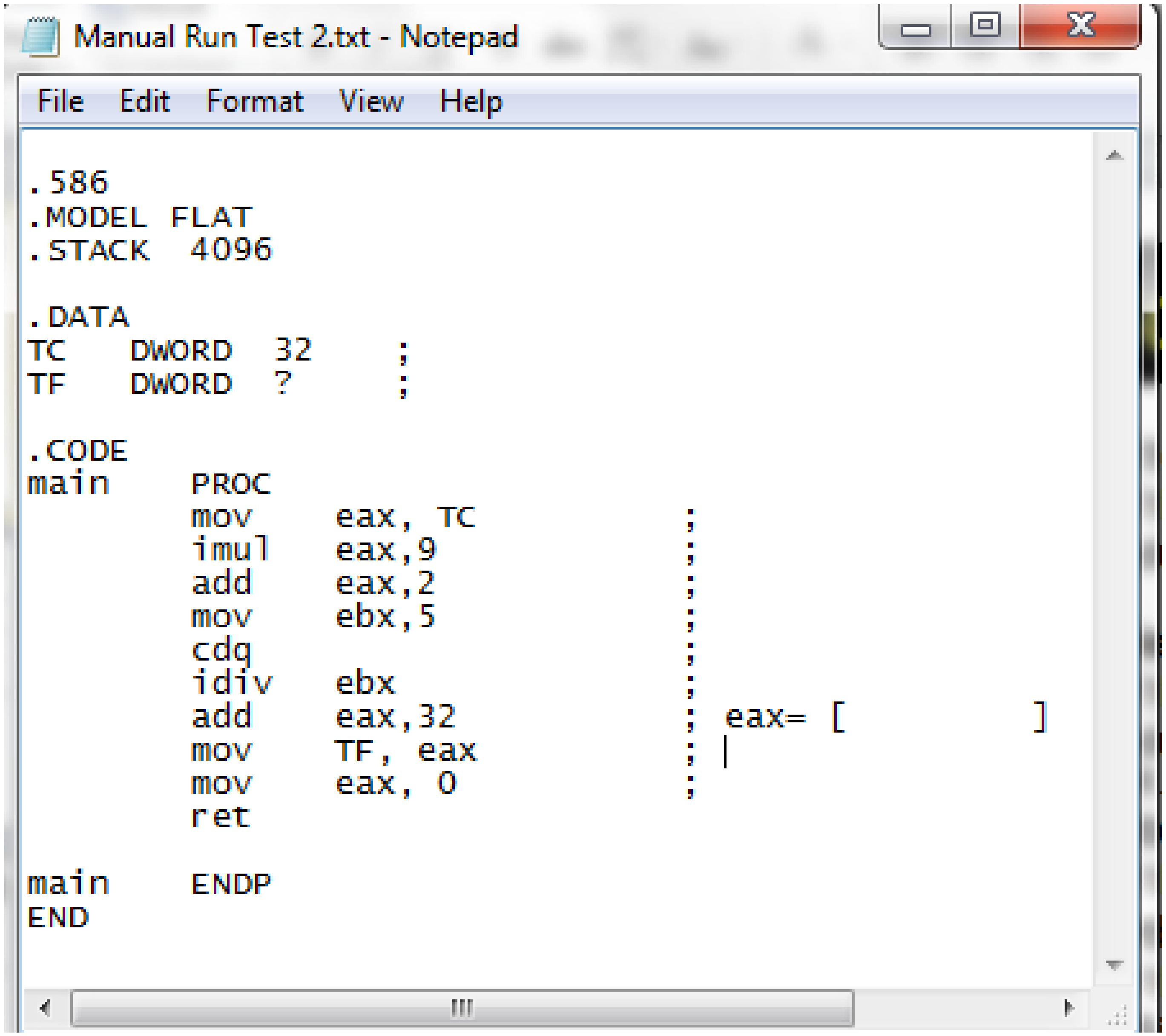
$-100/-13$ $\quad -100 = (-13)\cdot Q + R$ $\quad Q = 7,\ R = -9$

$100/13$ $\quad 100 = (13)\cdot Q + R$ $\quad Q = 7,\ R = 9$

DIV & IDIV Exercise

DIV Practice Exercise.docx

Manual Execution 2



```
.586
.MODEL FLAT
.STACK  4096

.DATA
TC     DWORD   32      ;
TF     DWORD   ?       ;

.CODE
main       PROC
           mov     eax, TC              ;
           imul    eax,9                ;
           add     eax,2                ;
           mov     ebx,5                ;
           cdq                          ;
           idiv    ebx                  ;
           add     eax,32               ; eax= [          ]
           mov     TF, eax              ; |
           mov     eax, 0               ;
           ret

main       ENDP
END
```

Integer

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

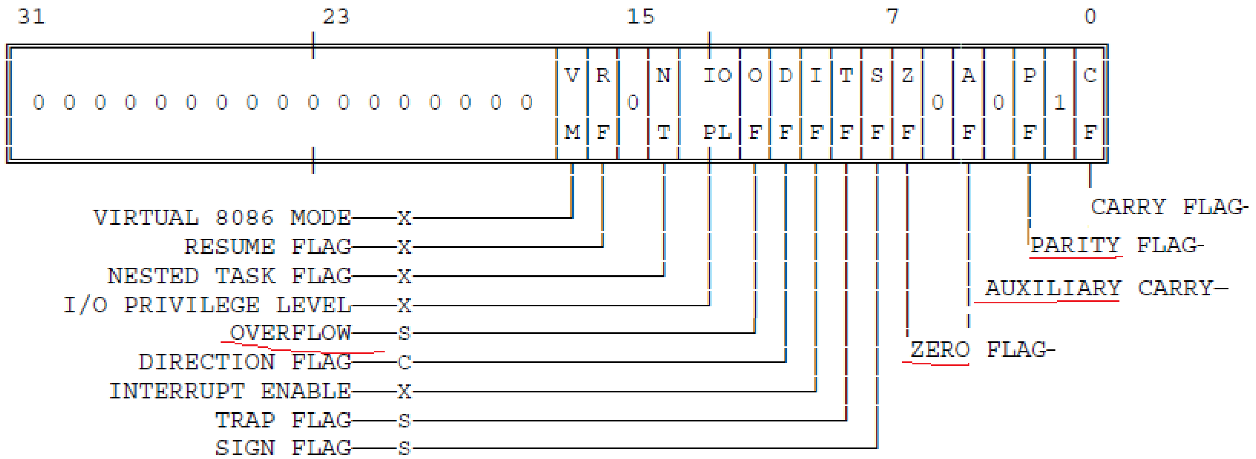
Round off $\rightarrow \textcircled{3}$

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

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

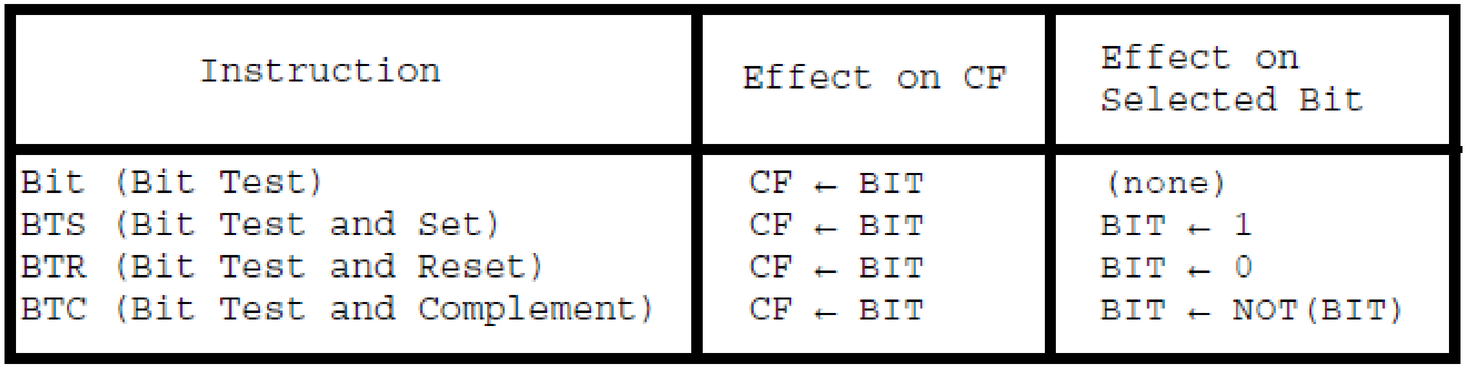
Boolean Operation Instruction

- 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       X     X      10001000100010001000100010001111
OR SAL

AFTER SHL        1     1 ←── 00010001000100010001000100011110 ←── 0
OR SAL BY 1

AFTER SHL        X     0 ←── 00100010001000100011110000000000 ←── 0
OR SAL BY 10
```

```
                                    OPERAND                        CF

BEFORE SHR             10001000100010001000100010001111           X

AFTER SHR       0─────►01000100010001000100010001000111─────►1
BY 1

AFTER SHR       0─────►00000000001000100010001000100010─────►O
BY 10
```

```
                                POSITIVE OPERAND                CF

BEFORE SAR             01000100010001000100010001000111          X

AFTER SAR       0─────►00100010001000100010001000100011─────►1
BY 1

                                NEGATIVE OPERAND                CF

BEFORE SAR             11000100010001000100010001000111          X

AFTER SAR       0─────►11100010001000100010001000100011─────►1
BY 1
```

Example

(Before) (After)

ecx : xxxx A9 D7

A 9 D 7

1010 1001 1101 0111

→ SAL CX, 1 0101 0011 1010 1110 ← 0

5 3 A E

eax : xxxx A9 D7

1010 1001 1101 0111

→ SHR AX, 1 0 → 0101 0100 1110 1011

5 4 E B

sign

$\boxed{1}$010 1001 1101 0111

bx : A9 D7

→ SAR bx, 1 1101 0100 1110 1011

D 4 E B

Example

```
dx : A9 D7                     1010  1001  1101  0111
                                      ────→
       → SHR  dx,4     0 →    0000  1010  1001  1101
                                 0     A     9     D

Ax : A9 D7                     1010  1001  1101  0111
CL :  04   → SAR  ax,CL                ────→
                          →    1101  0100  1110  1011  ①
                          →    1110  1010  0111  0101  ②
                          →    1111  0101  0011  1010  ③
                          →    1111  1010  1001  1101  ④
                                 F     A     9     D
```

Shift Practice 1

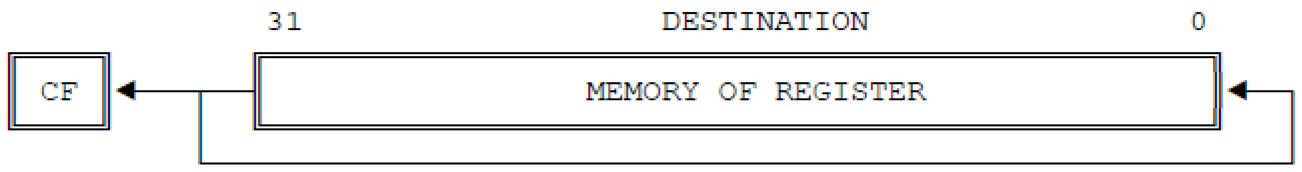
- (1) Before: [AX]=A8B5
  - Instruction: SHL AX,1
  - After: [AX]=
- (2) Before: [AX]=A8B5
  - Instruction: SHR AX,1
  - After: [AX]=
- (3) Before: [AX]=A8B5
  - Instruction: SAR AX,1
  - After: [AX]=

SHIFT Practice 2

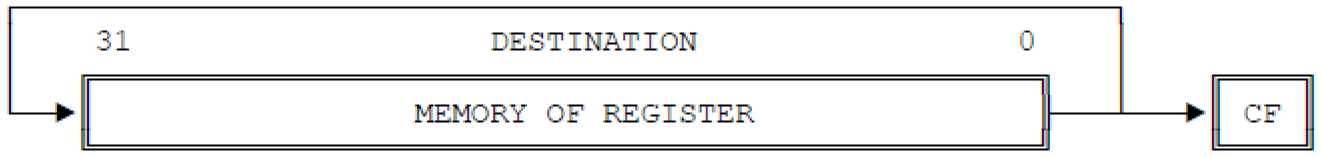
- (1) Before: [AX]=A8B5; [CL]=04
  - Instruction: SAL AX,CL
  - After: [AX]=
- (2) Before: [AX]=A8B5; [CL]=04
  - Instruction: SAR AX,CL
  - After: [AX]=
- (3) Before: [AX]=A8B5
  - Instruction: SHR AX,4
  - After: [AX]=

Rotation

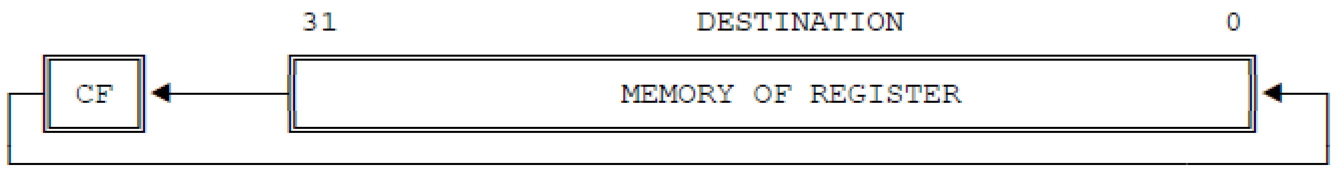
ROL



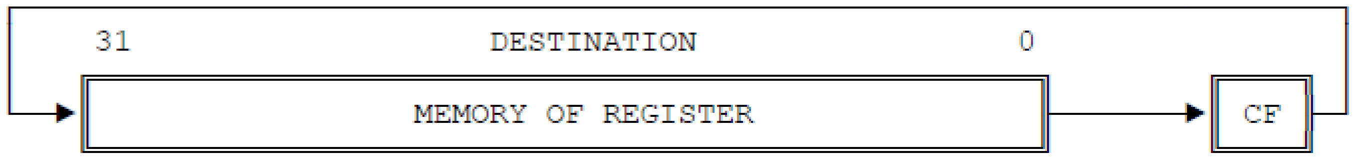
ROR



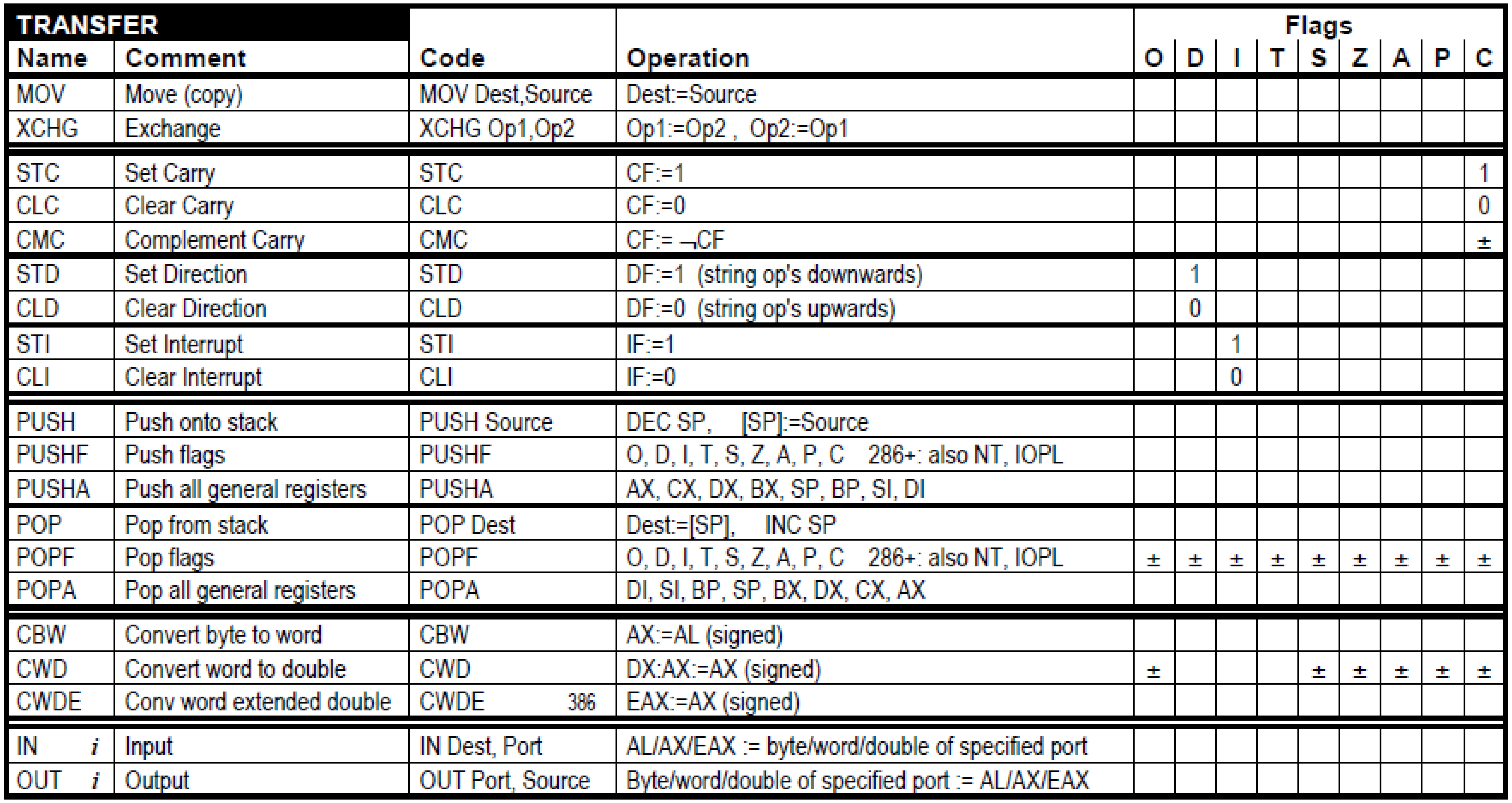
RCL



RCR



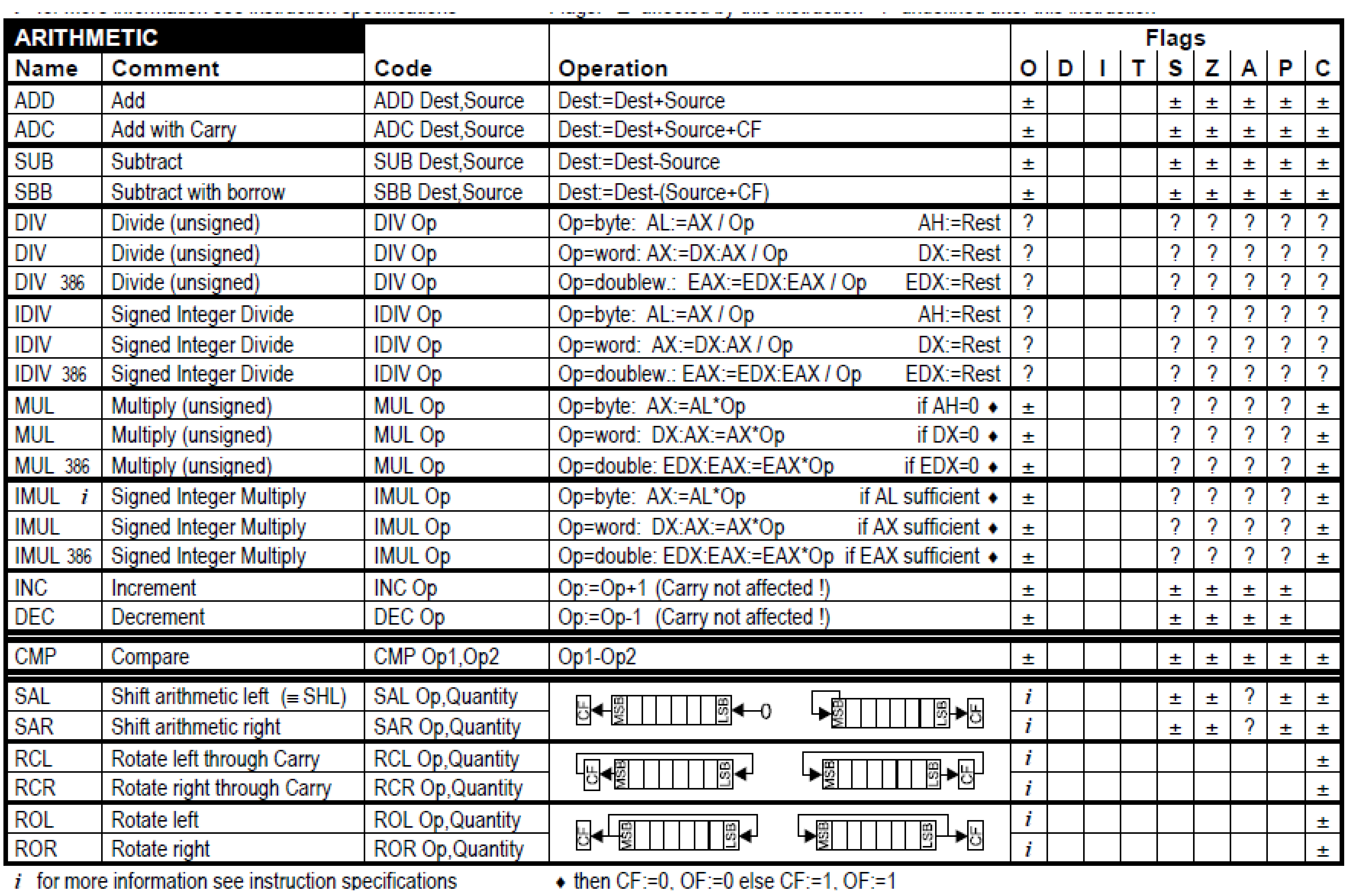
MOVE

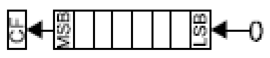
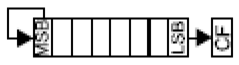
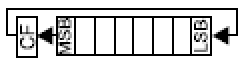
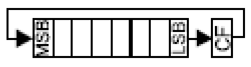
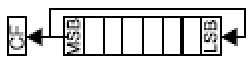
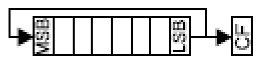


TRANSFER				**Flags**								
Name	**Comment**	**Code**	**Operation**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									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

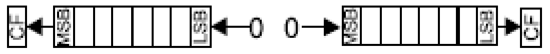
Arithmetic



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*								±

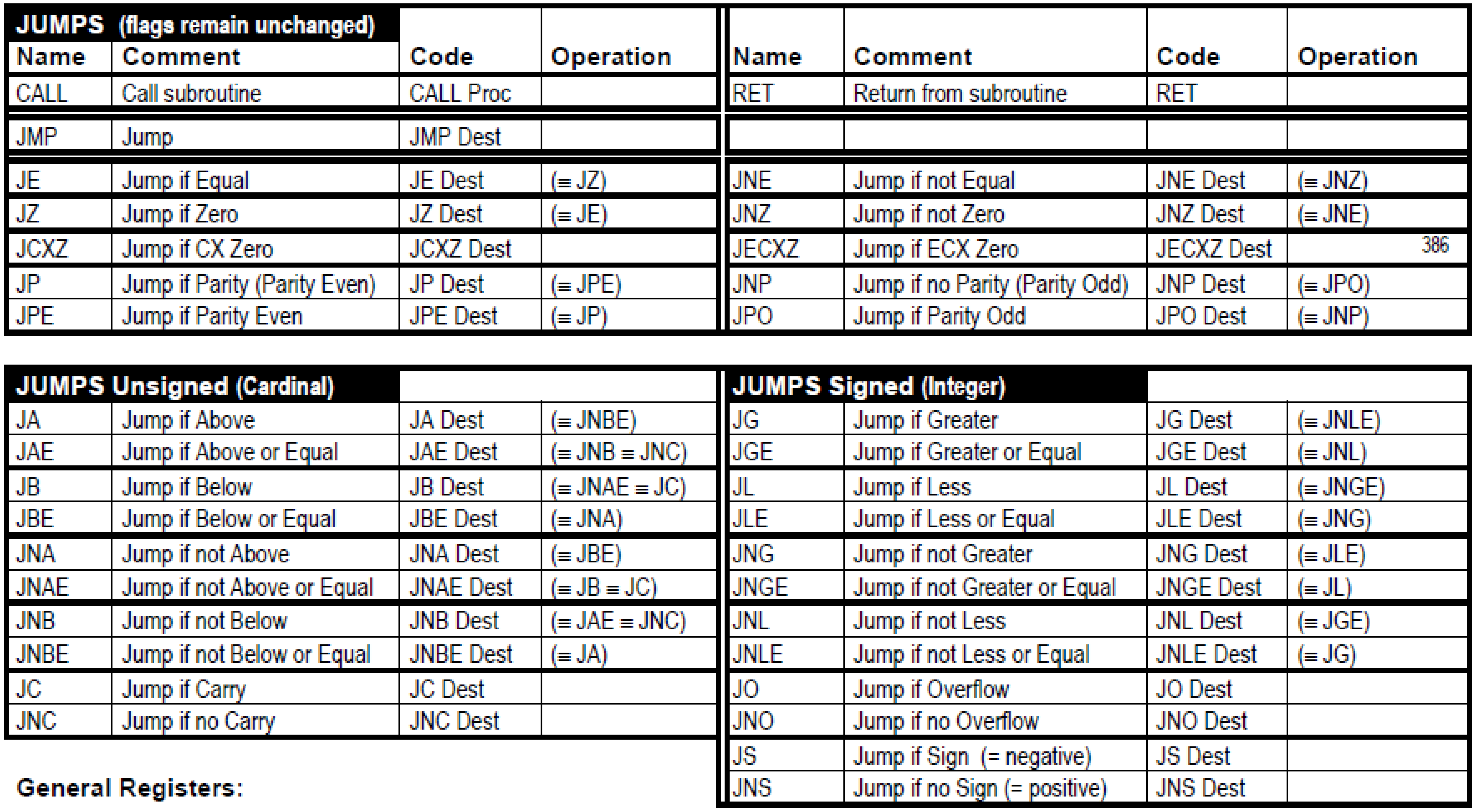
i for more information see instruction specifications ♦ then CF:=0, OF:=0 else CF:=1, OF:=1

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	Name	Comment	Code	Operation
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest	(≡ JZ)	JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)	JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	386
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)	JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)	JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

JUMPS Unsigned (Cardinal)				JUMPS Signed (Integer)			
JA	Jump if Above	JA Dest	(≡ JNBE)	JG	Jump if Greater	JG Dest	(≡ JNLE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)	JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)	JL	Jump if Less	JL Dest	(≡ JNGE)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)	JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNA	Jump if not Above	JNA Dest	(≡ JBE)	JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)	JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)	JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)	JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JC	Jump if Carry	JC Dest		JO	Jump if Overflow	JO Dest	
JNC	Jump if no Carry	JNC 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: