x86 Assembly Programming Part 2

EECE416 uC

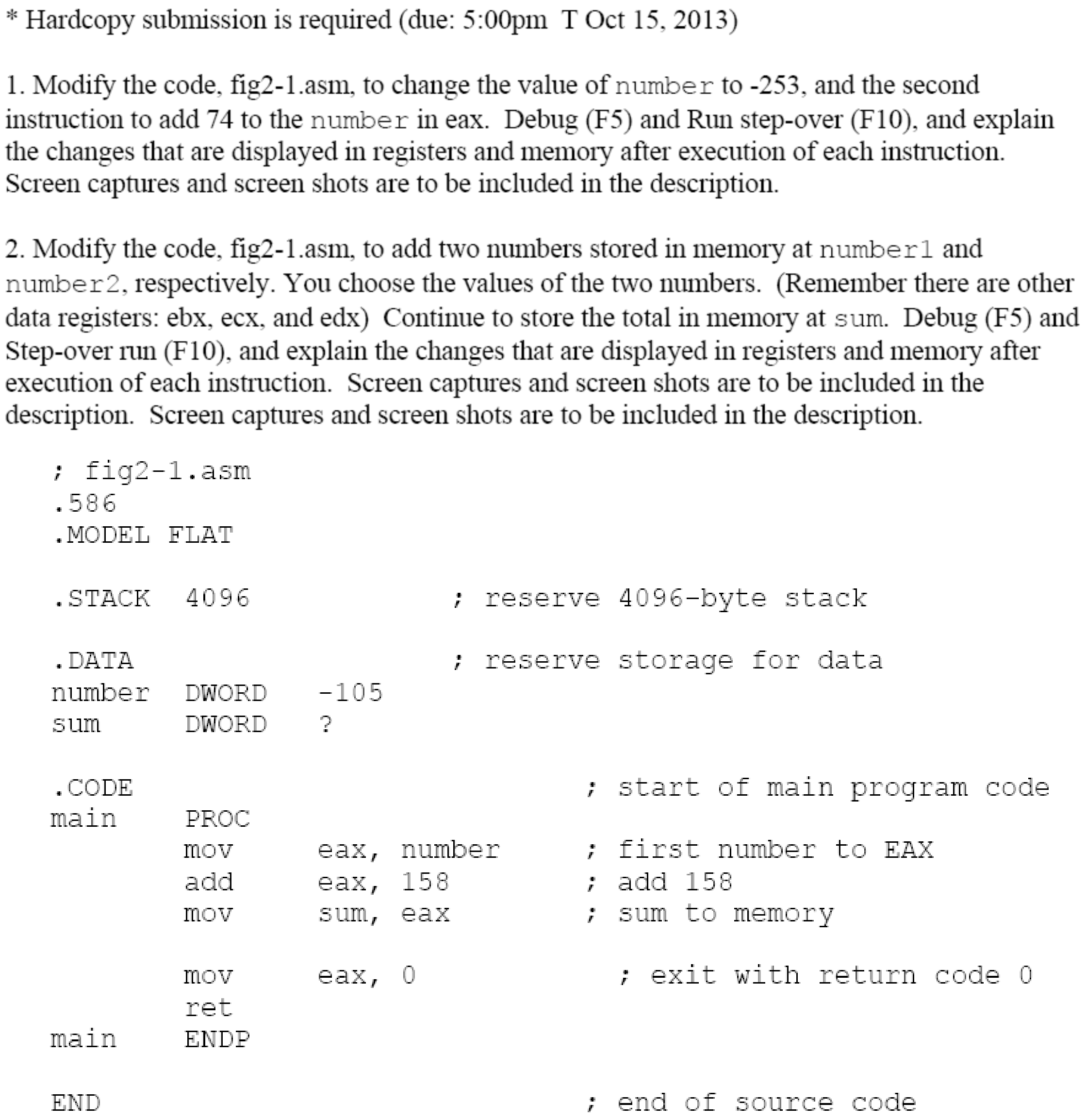
Charles Kim
Howard University

Resources:

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

WWW.MWFTR.COM

Reminder – Coding Assignment



* Hardcopy submission is required (due: 5:00pm T Oct 15, 2013)

1. Modify the code, fig2-1.asm, to change the value of `number` to -253, and the second instruction to add 74 to the `number` in eax. Debug (F5) and Run step-over (F10), and explain the changes that are displayed in registers and memory after execution of each instruction. Screen captures and screen shots are to be included in the description.

2. Modify the code, fig2-1.asm, to add two numbers stored in memory at `number1` and `number2`, respectively. You choose the values of the two numbers. (Remember there are other data registers: ebx, ecx, and edx) Continue to store the total in memory at `sum`. Debug (F5) and Step-over run (F10), and explain the changes that are displayed in registers and memory after execution of each instruction. Screen captures and screen shots are to be included in the description. Screen captures and screen shots are to be included in the description.

```
; fig2-1.asm
.586
.MODEL FLAT

.STACK  4096            ; reserve 4096-byte stack

.DATA                   ; reserve storage for data
number  DWORD   -105
sum     DWORD   ?

.CODE                           ; start of main program code
main    PROC
        mov     eax, number     ; first number to EAX
        add     eax, 158        ; add 158
        mov     sum, eax        ; sum to memory

        mov     eax, 0            ; exit with return code 0
        ret
main    ENDP

END                             ; end of source code
```

Listing (.LST) File of Assembly Code (.asm)

Example1.asm

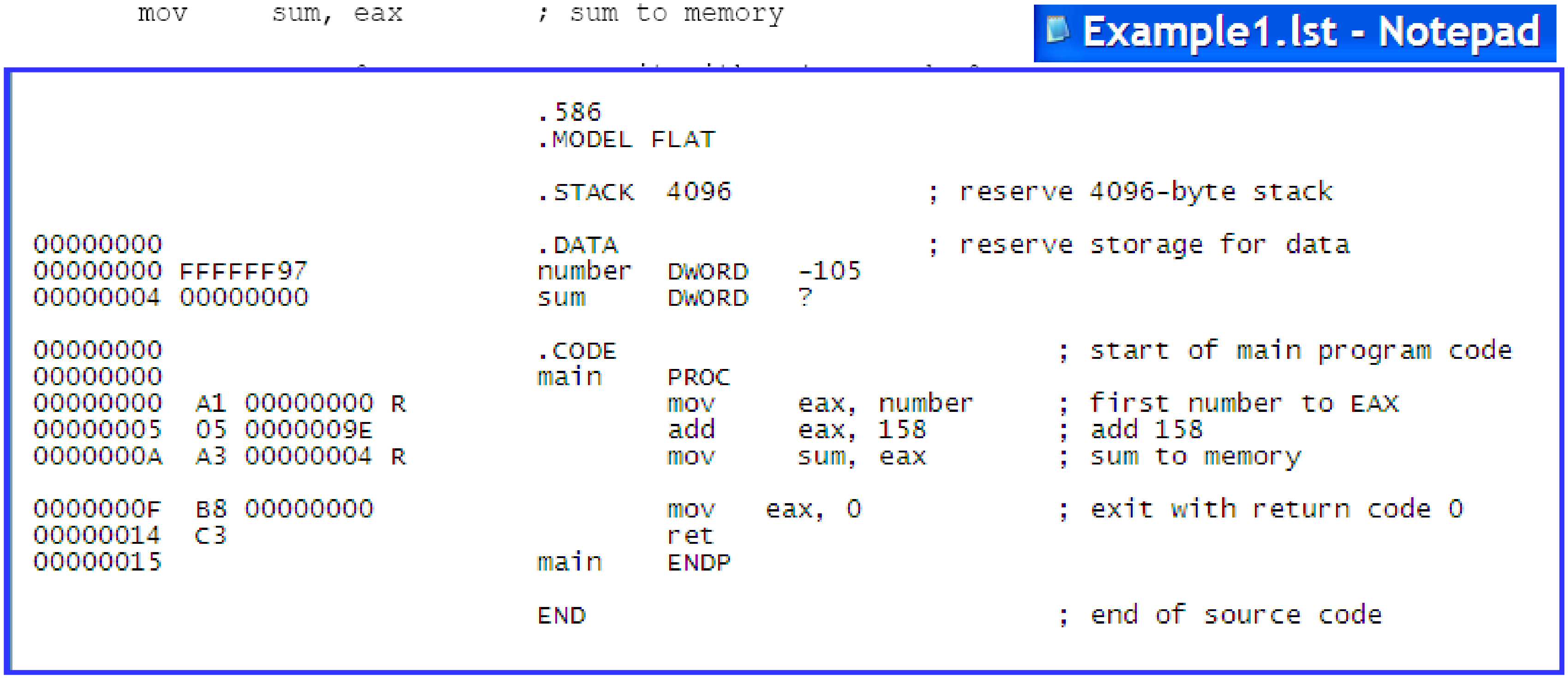
```
.586
.MODEL FLAT

.STACK  4096              ; reserve 4096-byte stack

.DATA                     ; reserve storage for data
number  DWORD   -105
sum     DWORD   ?

.CODE                               ; start of main program code
main    PROC
        mov     eax, number         ; first number to EAX
        add     eax, 158            ; add 158
        mov     sum, eax            ; sum to memory
```

.LST file is located inside the console32 sub-folder, inside the main console32 folder



Example1.lst - Notepad

```
                                    .586
                                    .MODEL FLAT

                                    .STACK  4096            ; reserve 4096-byte stack

 00000000                           .DATA                   ; reserve storage for data
 00000000 FFFFFF97                  number  DWORD   -105
 00000004 00000000                  sum     DWORD   ?

 00000000                           .CODE                           ; start of main program code
 00000000                           main    PROC
 00000000  A1 00000000 R                    mov     eax, number     ; first number to EAX
 00000005  05 0000009E                      add     eax, 158        ; add 158
 0000000A  A3 00000004 R                    mov     sum, eax        ; sum to memory

 0000000F  B8 00000000                      mov    eax, 0           ; exit with return code 0
 00000014  C3                               ret
 00000015                           main    ENDP

                                    END                             ; end of source code
```

Registers for x86

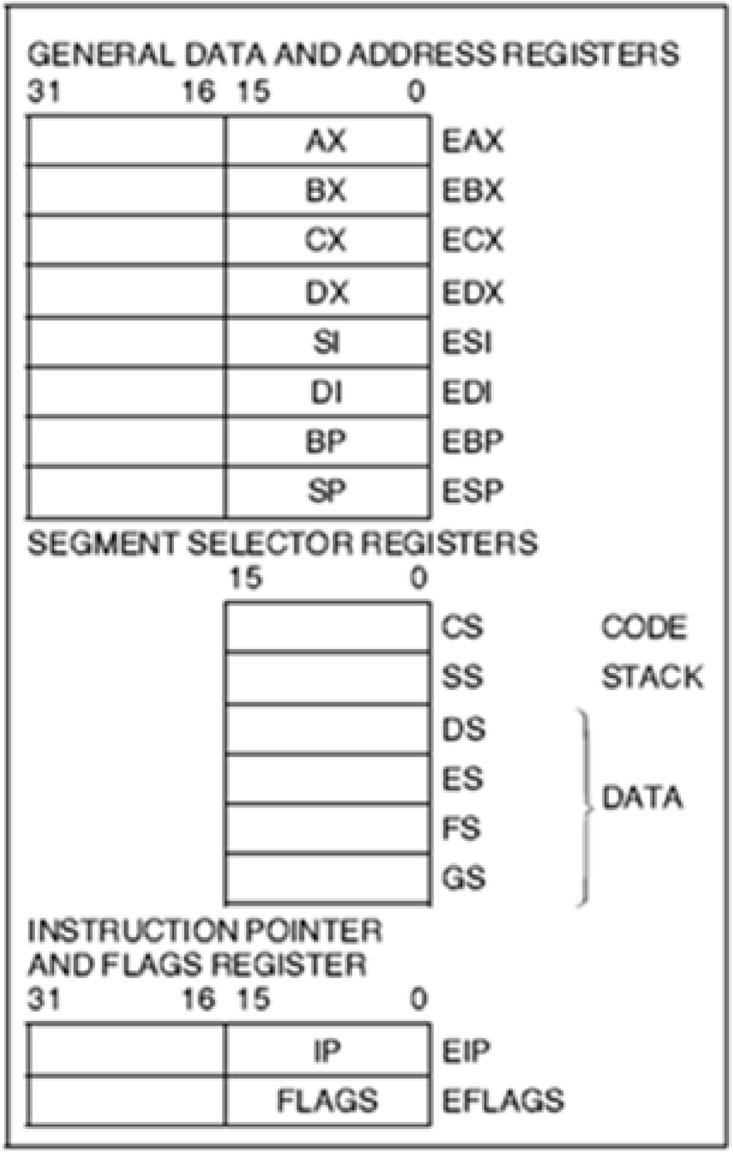
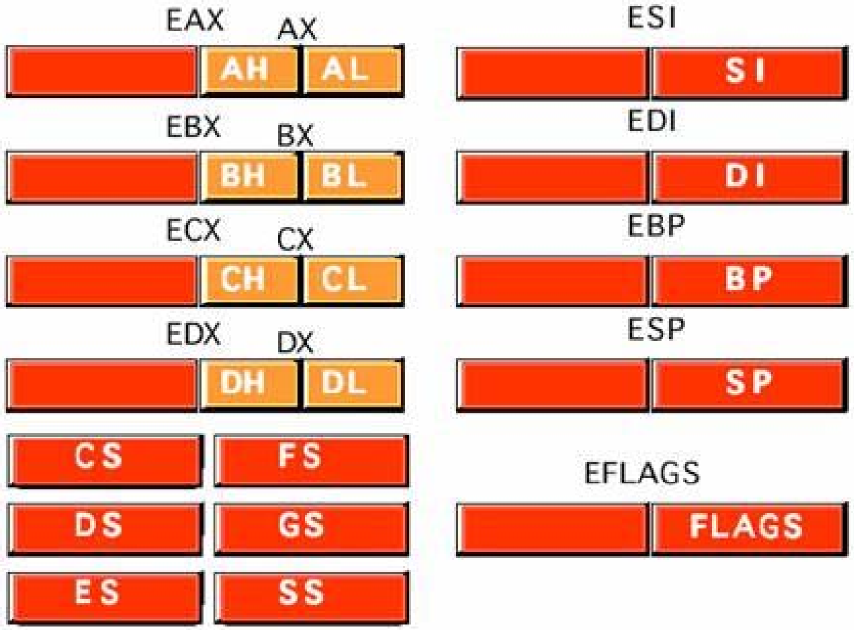
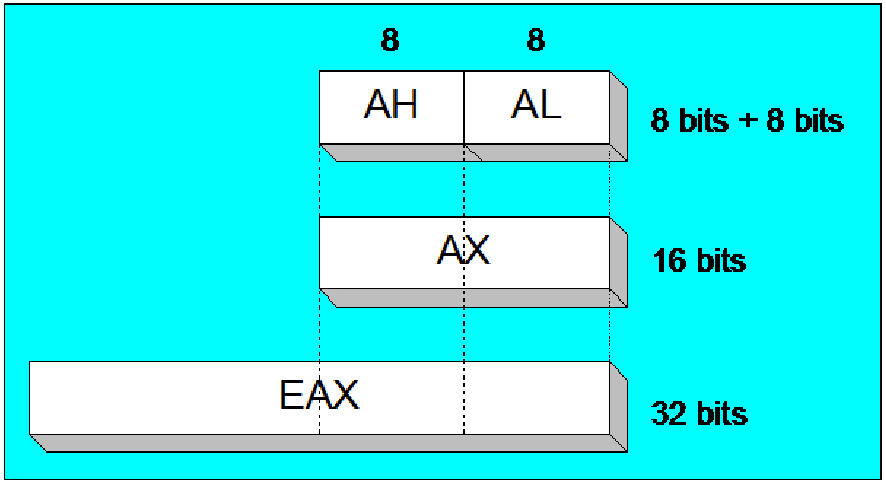


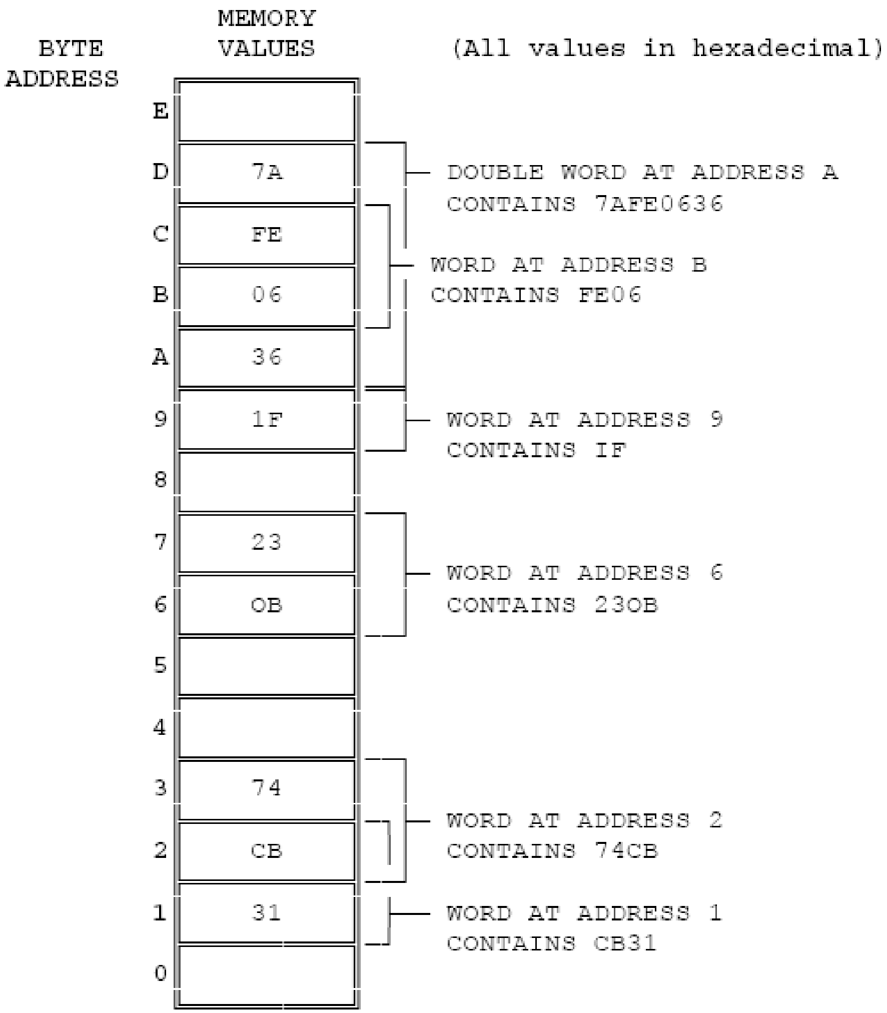
Figure 2-1. Intel386™ DX Base Architecture Registers

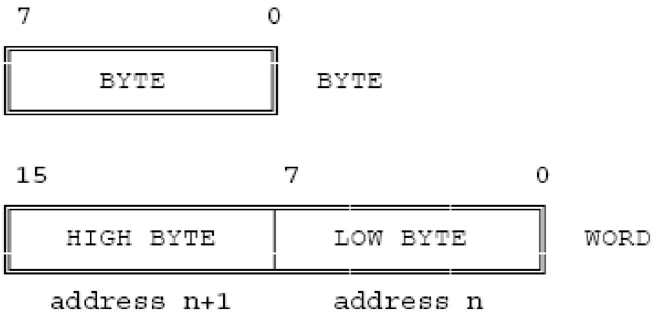


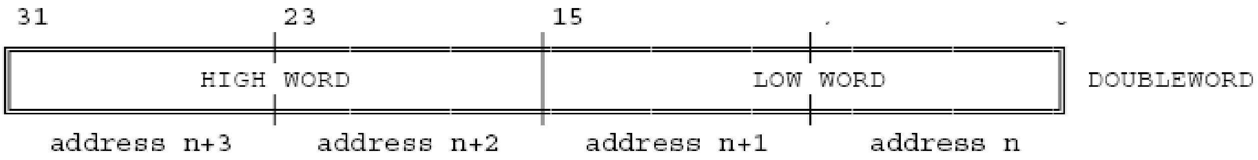


Basic Data Types

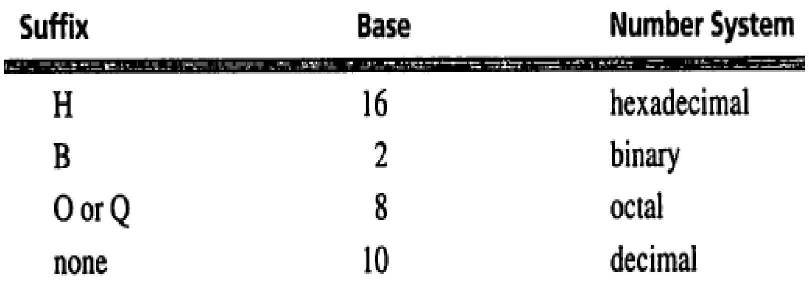
- Byte, Words (`WORD`), Double Words (`DWORD`)
- Little-Endian
- Align by 2 (`word`) or 4 (`Dword`) for better performance – instead of odd address


MEMORY VALUES
BYTE ADDRESS
(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



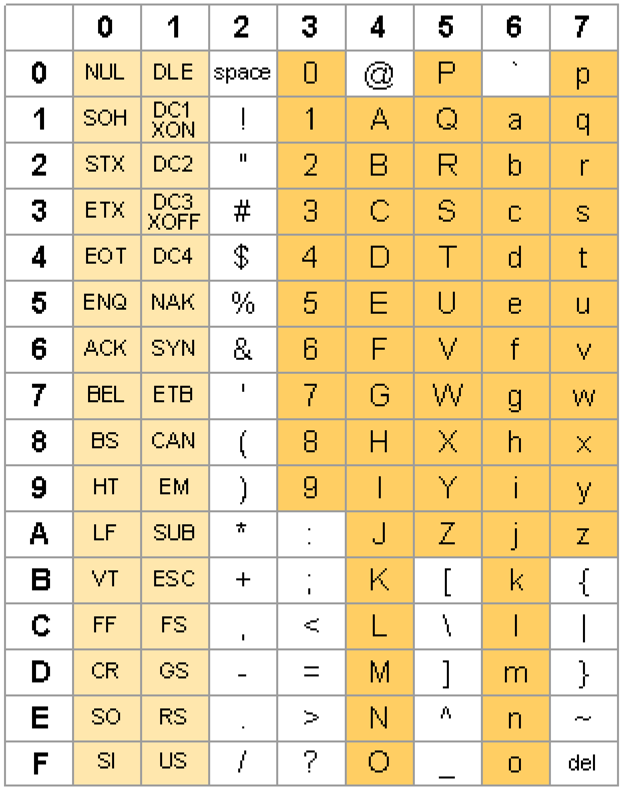


Data Declaration



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

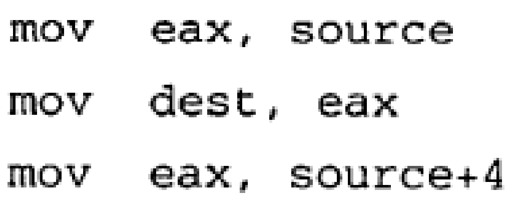
- Directives for Data Declaration and Reservation of Memory
  - BYTE: Reserves 1 byte in memory
    - Example: **D1 BYTE 20**
      D2 BYTE 00010100b
      String1 BYTE “Joe” ; [4A 6F 65]
  - WORD: 2 bytes are reserved
    - Example: **num1 WORD -10**
      num2 WORD FFFFH
  - DWORD: 4 bytes are reserved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



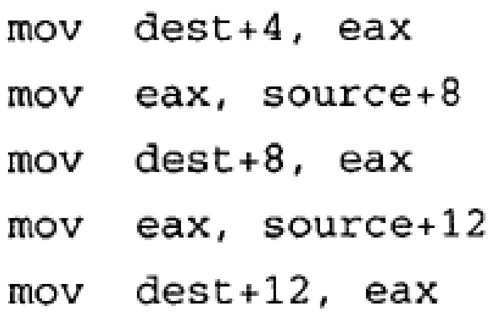
	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L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

Instruction Format

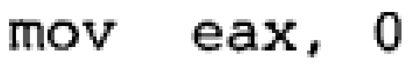
- Opcode:
  - specifies the operation performed by the instruction.
- Register specifier
  - an instruction may specify one or two register operands.
- Addressing-mode specifier
  - when present, specifies whether an operand is a register or memory location.
- Displacement
  - when the addressing-mode 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
```



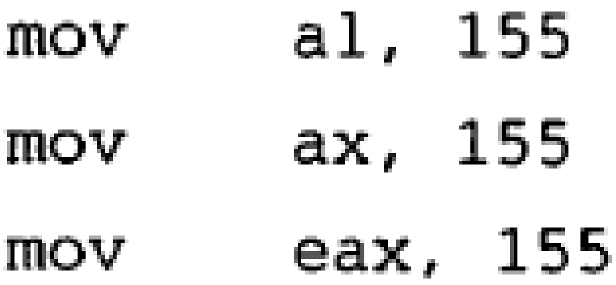
```
mov   dest+4, eax
mov   eax, source+8
mov   dest+8, eax
mov   eax, source+12
mov   dest+12, eax
```



```
mov   eax, 0
```

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          eax:[                    ]
mov     ax, 155          eax:[                    ]
mov     eax, 155         eax:[                    ]
```

- Further Example
- EAX: [01010101] ← before

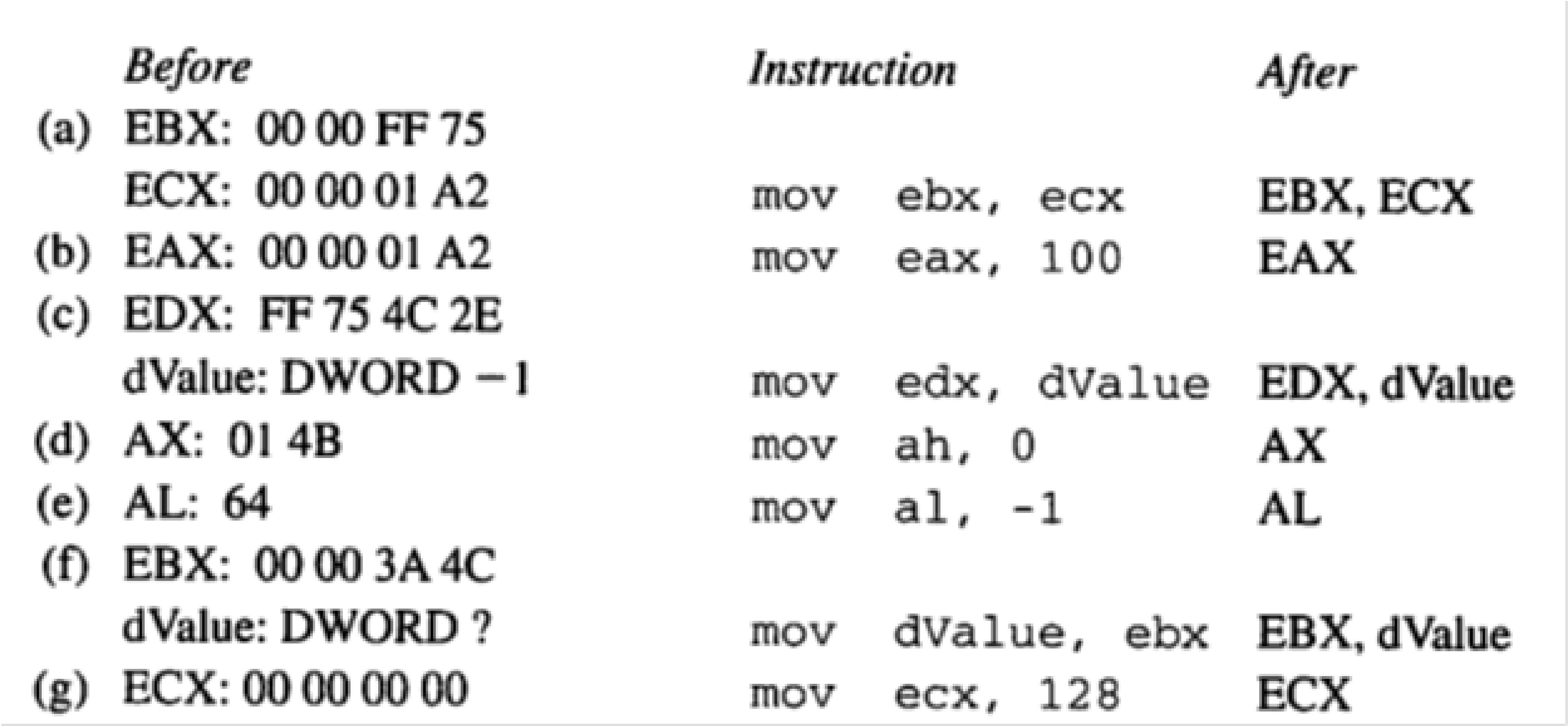
```
mov al -10 ;    EAX:[                    ]
mov ax, -10;    EAX: [                   ]
mov eax, -10;   EAX: [                 ]
```

Exercise of Register Size and Data

- 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

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

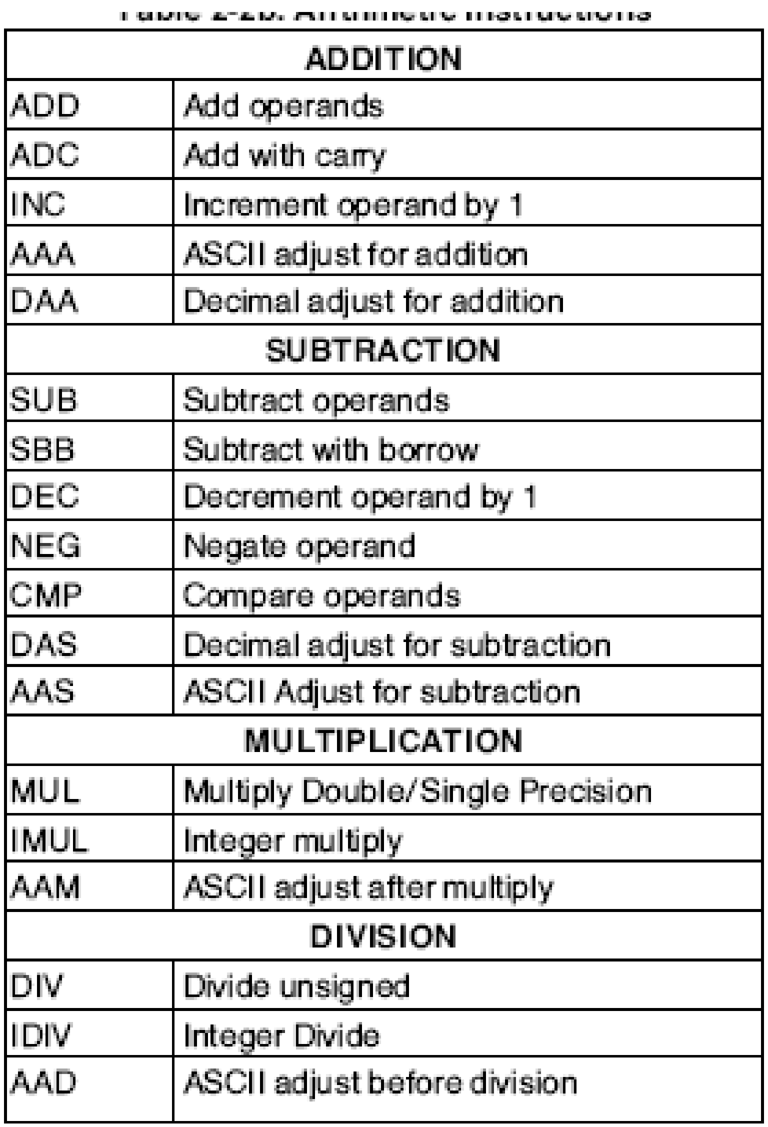


Table 2-2b. Arithmetic Instructions

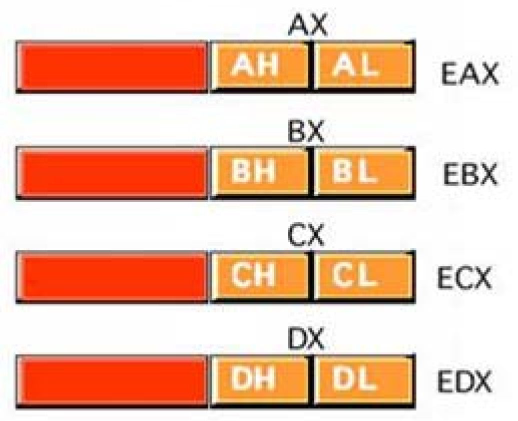
ADDITION	
ADD	Add operands
ADC	Add with carry
INC	Increment operand by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
SUBTRACTION	
SUB	Subtract operands
SBB	Subtract with borrow
DEC	Decrement operand by 1
NEG	Negate operand
CMP	Compare operands
DAS	Decimal adjust for subtraction
AAS	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

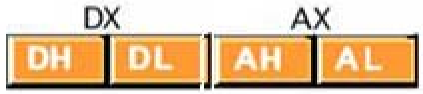
Data movement Instructions

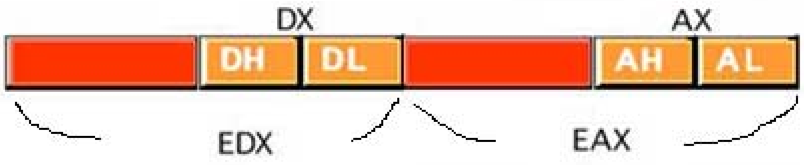
- MOV (Move)
  - transfers a byte, word, or doubleword from the source operand to the destination operand: R→ M. M → R, R→ R, I→R, I→ M
  - The MOV instruction cannot move M→M or from SR → SR (segment register)
  - M→M via MOVS (string)
- XCHG (Exchange)
  - swaps the contents of two operands.
  - swap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

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.
- 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.

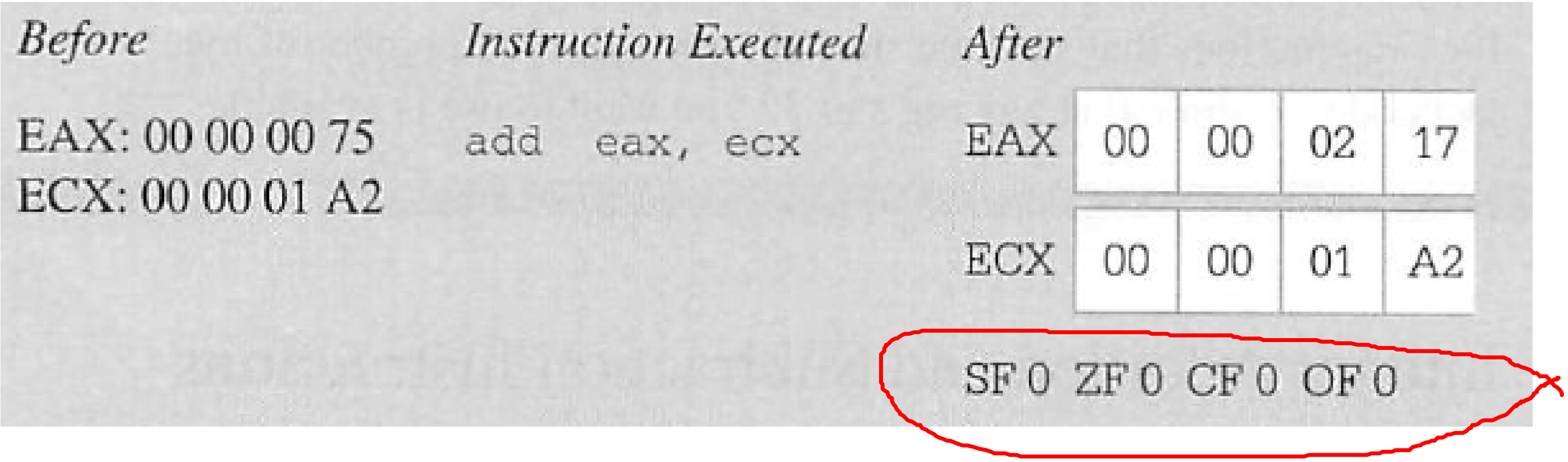






Addition Instruction

- ADD (Add Integers)
  - (DST + SRC) → DST
  - replaces the destination operand with the sum of the source and destination operands. OF, SF, ZF, CF are all affected.
- ADC (Add Integers with Carry)
  - (DST + SRC + 1) → DST (if CF=1)
  - (DST + SRC) → DST (if CF=0)
  - 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.

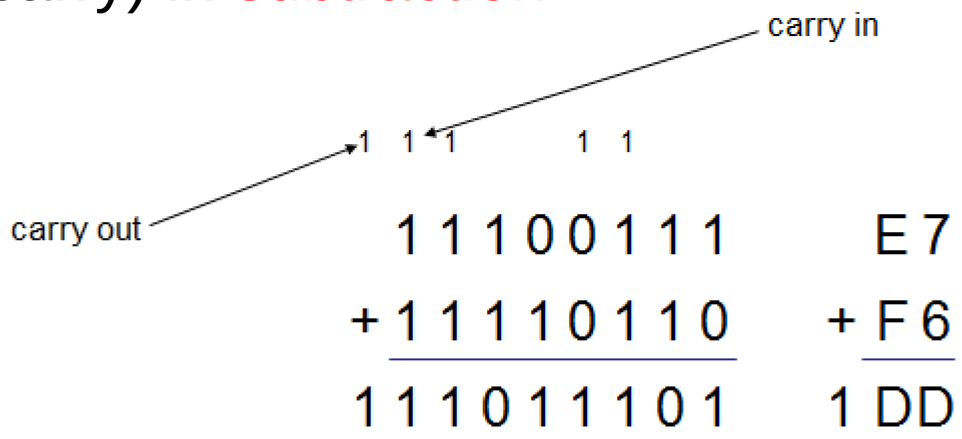


Before	*Instruction Executed*	*After*				
EAX: 00 00 00 75	`add eax, ecx`	EAX	00	00	02	17
ECX: 00 00 01 A2		ECX	00	00	01	A2
		SF 0 ZF 0 CF 0 OF 0				

- label mnemonic dst, src

Flags

- **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 high-order (sign, leftmost) bit position during addition is called a carry in.
  - CF=1 for borrow (or no carry) in subtraction.



Flags

- **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

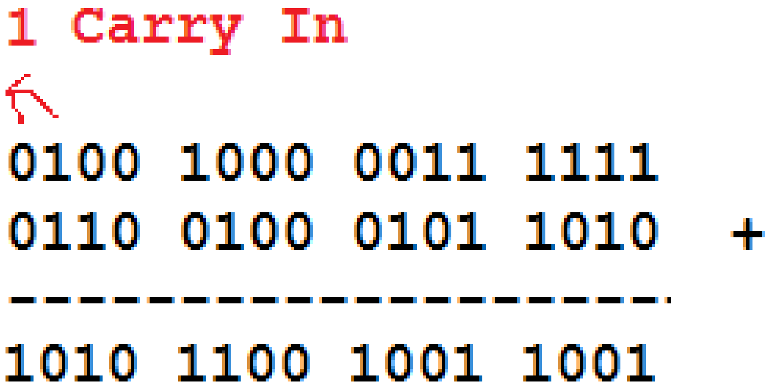
- Example

  483F + 645A → AC99

  Carry In but no Carry Out

  → OF=1

  No Carry Out → CF=0



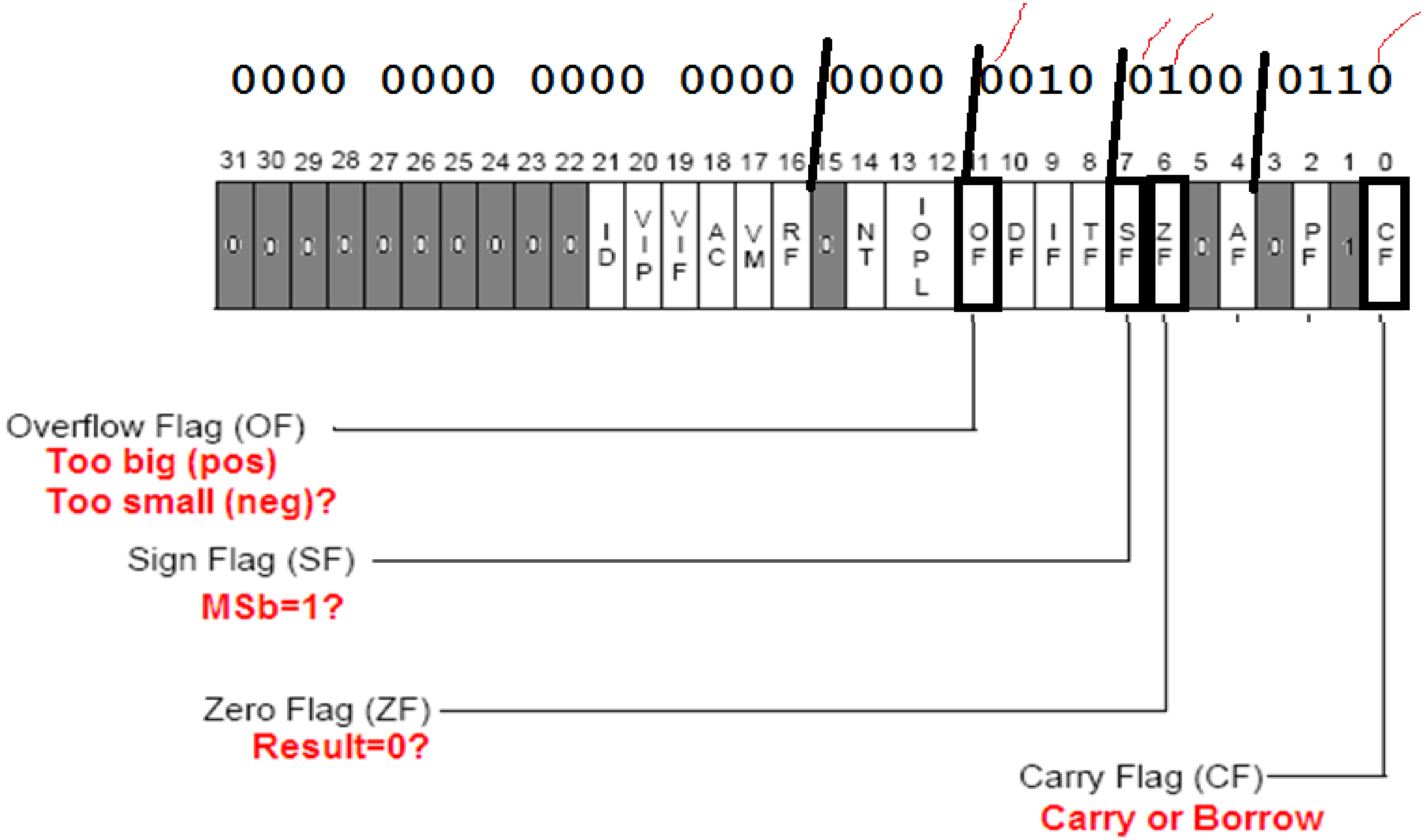
```
1 Carry In

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

Status Flags in Console32

- EFL=00 00 02 46 (after `mov eax, number`)



0000 0000 0000 0000 0000 0010 0100 0110

Overflow Flag (OF)
Too big (pos)
Too small (neg)?

Sign Flag (SF)
MSb=1?

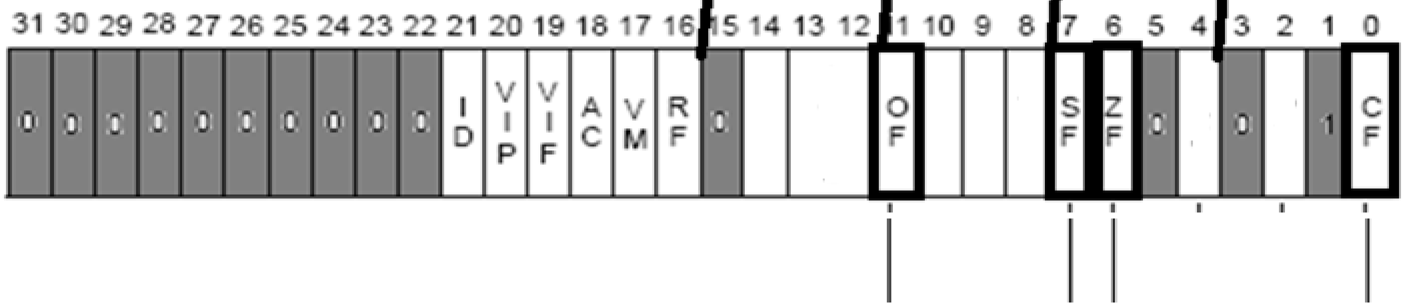
Zero Flag (ZF)
Result=0?

Carry Flag (CF)
Carry or Borrow

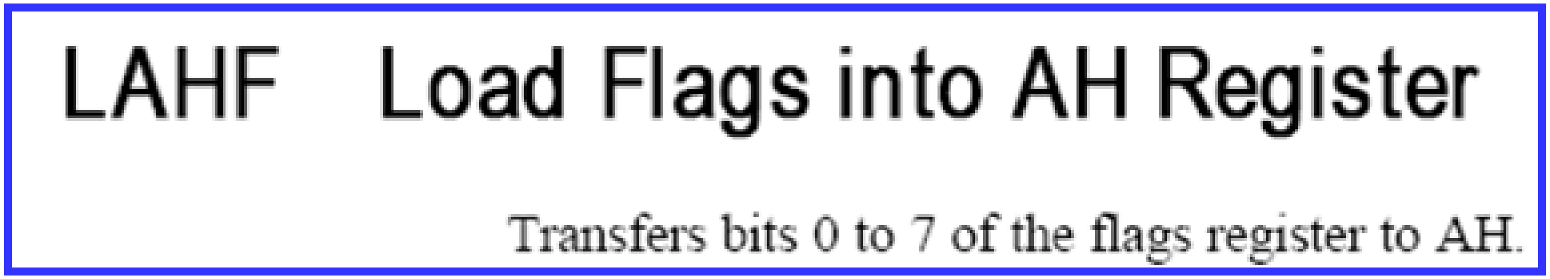
Status Flags in Console32

- EFL=00 00 02 17 (after `add eax, 158`)

```
0000 0000 0000 0000 0000 0010 0001 0111
```



```
               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
```

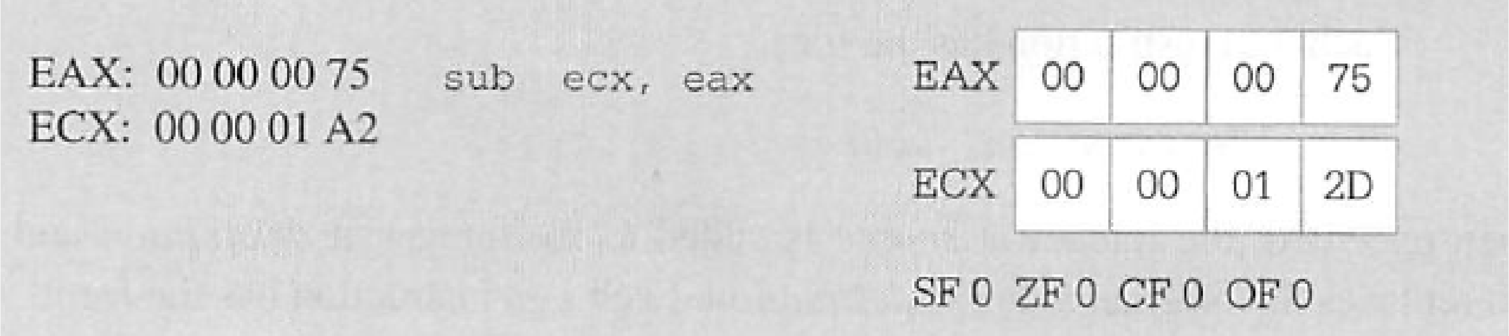


LAHF Load Flags into AH Register

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

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.
- SBB (Subtract Integers with Borrow)
  - DST – SRC (if CF=0)
  - DST – 1 (if CF=1)
  - 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.



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

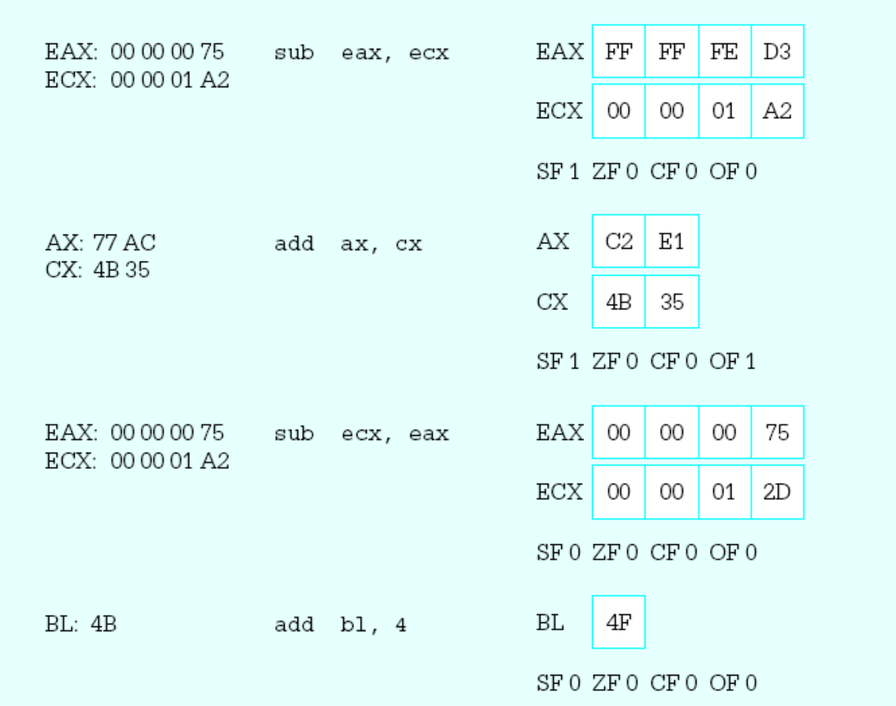
```
sub   ecx, eax
```

EAX	00	00	00	75
ECX	00	00	01	2D

SF 0 ZF 0 CF 0 OF 0

- label mnemonic dst, src

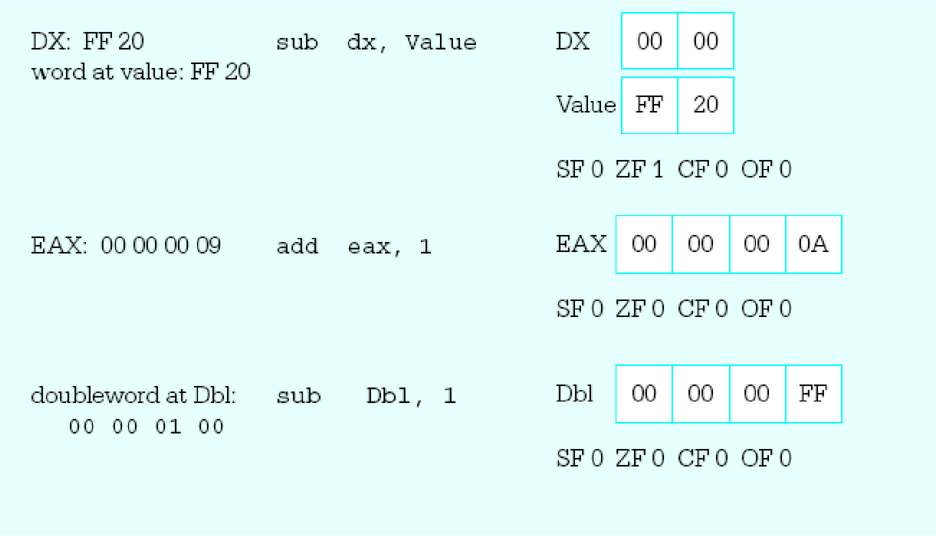
ADD & SUB Examples



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

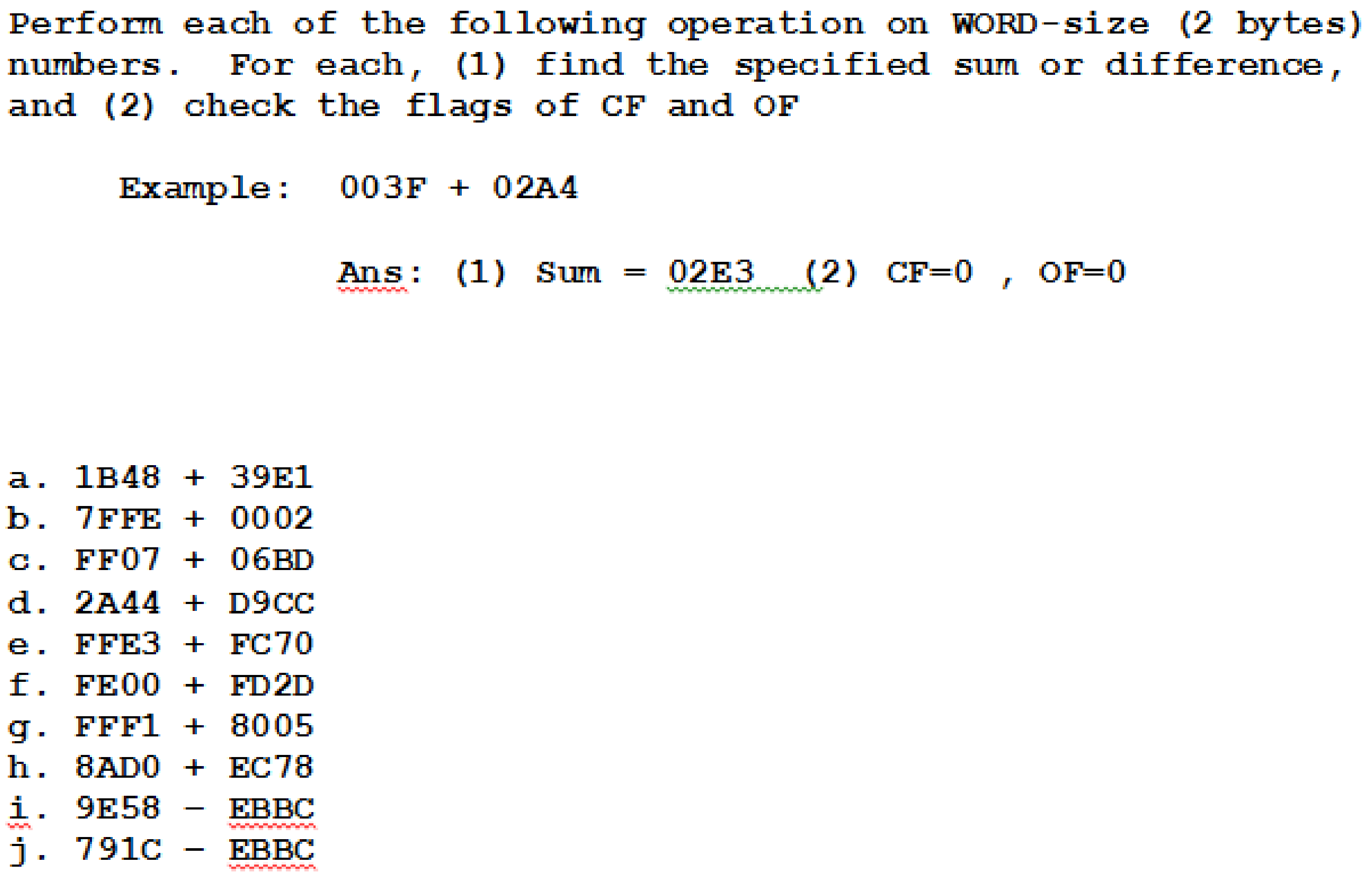
- SUB [dst] – [src]

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



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

Add , Sub, and Flag Practice



Perform each of the following operation on WORD-size (2 bytes) numbers. For each, (1) find the specified sum or difference, and (2) check the flags of CF and OF

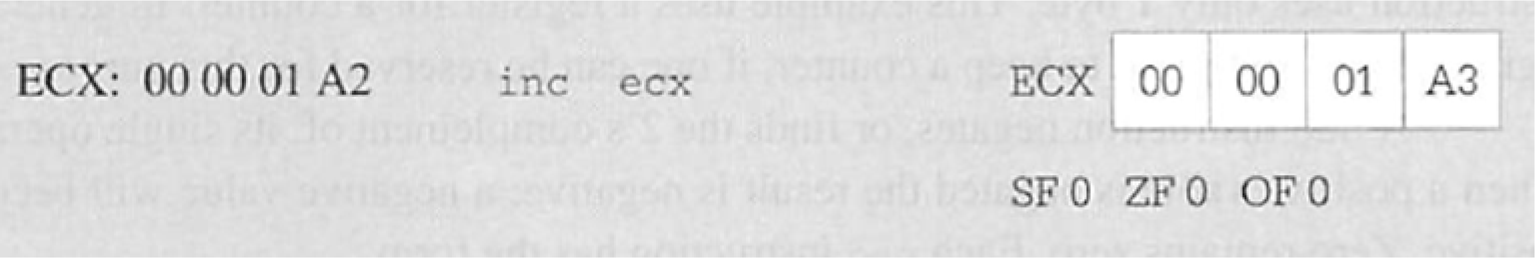
Example: 003F + 02A4

Ans: (1) Sum = 02E3 (2) CF=0 , OF=0

a. 1B48 + 39E1
b. 7FFE + 0002
c. FF07 + 06BD
d. 2A44 + D9CC
e. FFE3 + FC70
f. FE00 + FD2D
g. FFF1 + 8005
h. 8AD0 + EC78
i. 9E58 – EBBC
j. 791C – EBBC

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.

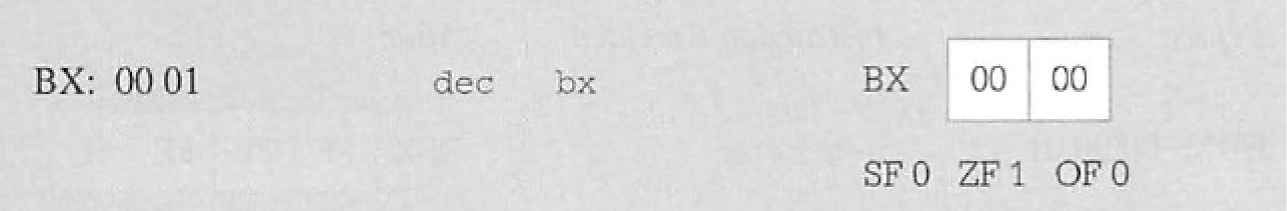


ECX: 00 00 01 A2 inc ecx

ECX | 00 | 00 | 01 | A3

SF 0 ZF 0 OF 0

- 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.

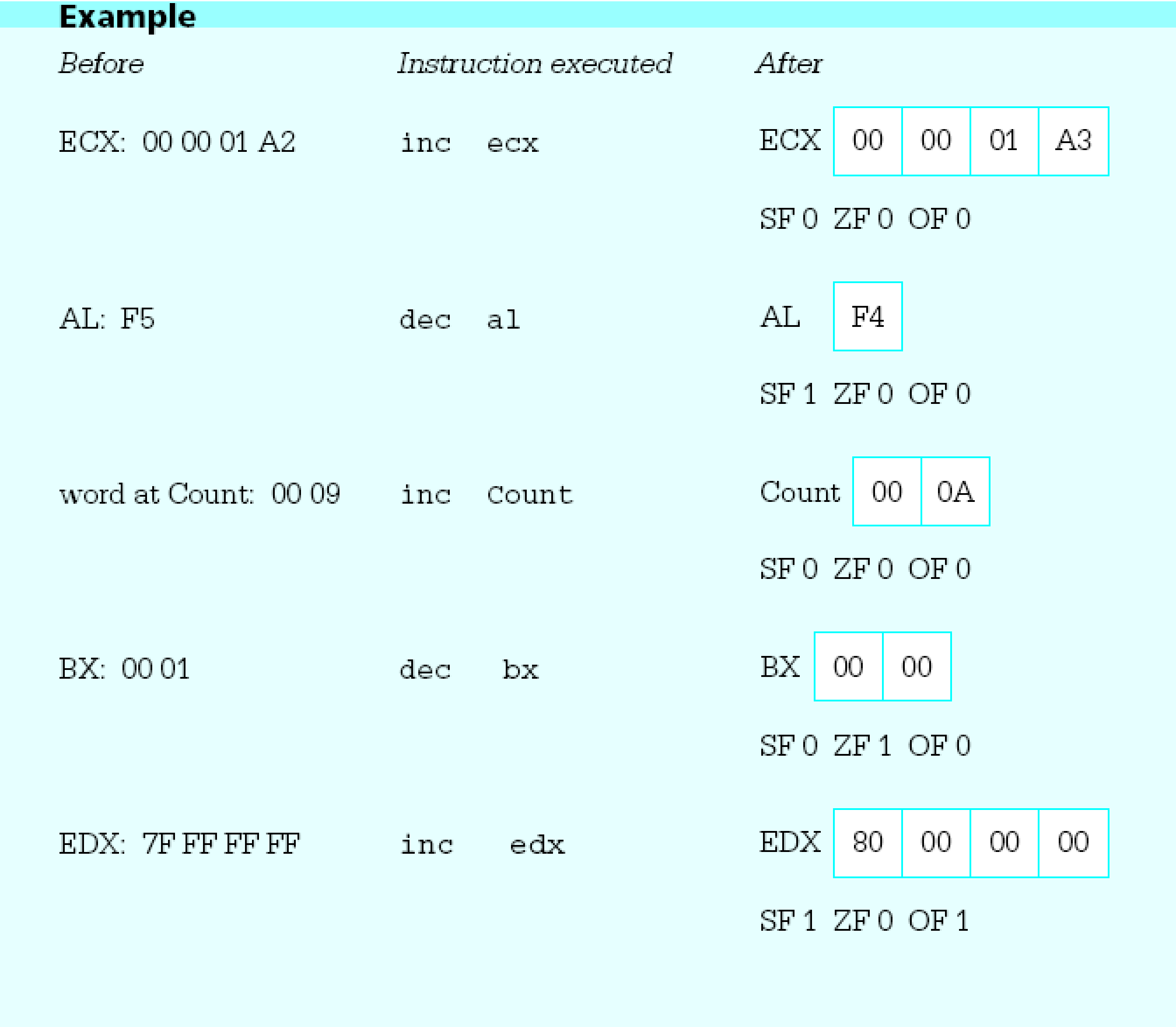


BX: 00 01 dec bx

BX | 00 | 00

SF 0 ZF 1 OF 0

INC + DEC examples

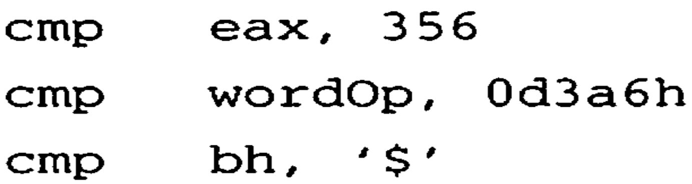


Example

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

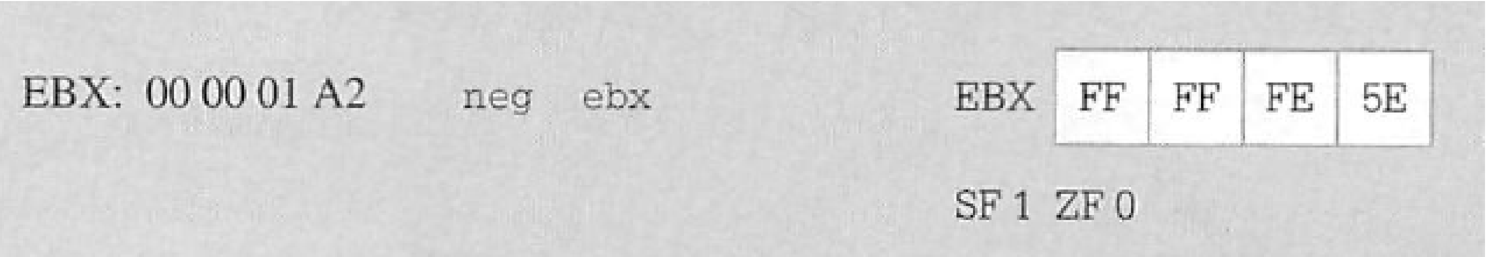
CMP + NEG

- 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, '$'
```

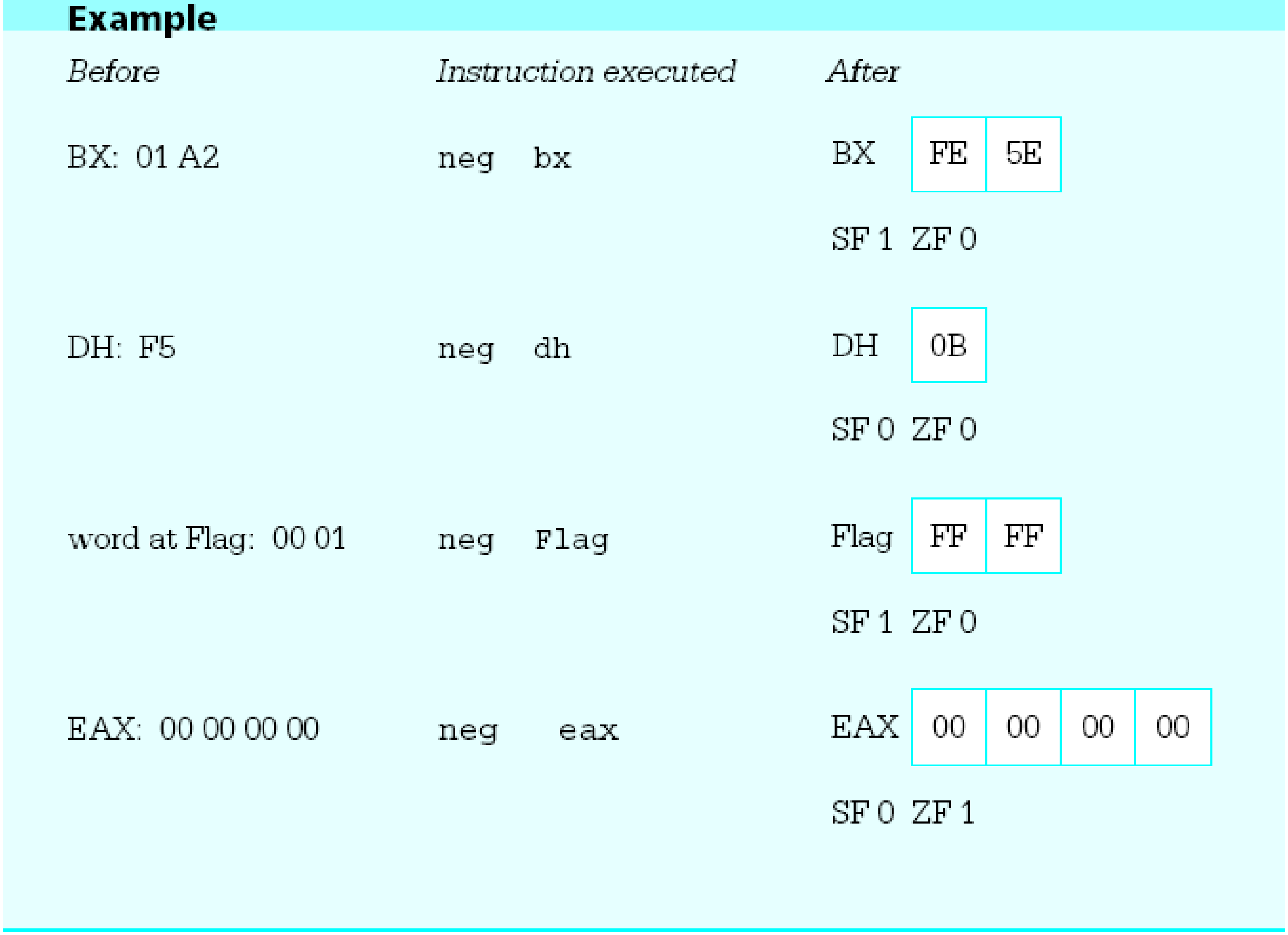
- NEG (Negate)
  - **0 – DST → DST**
  - **subtracts a signed integer operand from zero**. The effect of NEG is to **reverse the sign** of the operand from positive to negative or from negative to positive (i.e., **16's complement**)
  - **SF and ZF are affected**



EBX: 00 00 01 A2   neg ebx   EBX FF FF FE 5E

SF 1 ZF 0

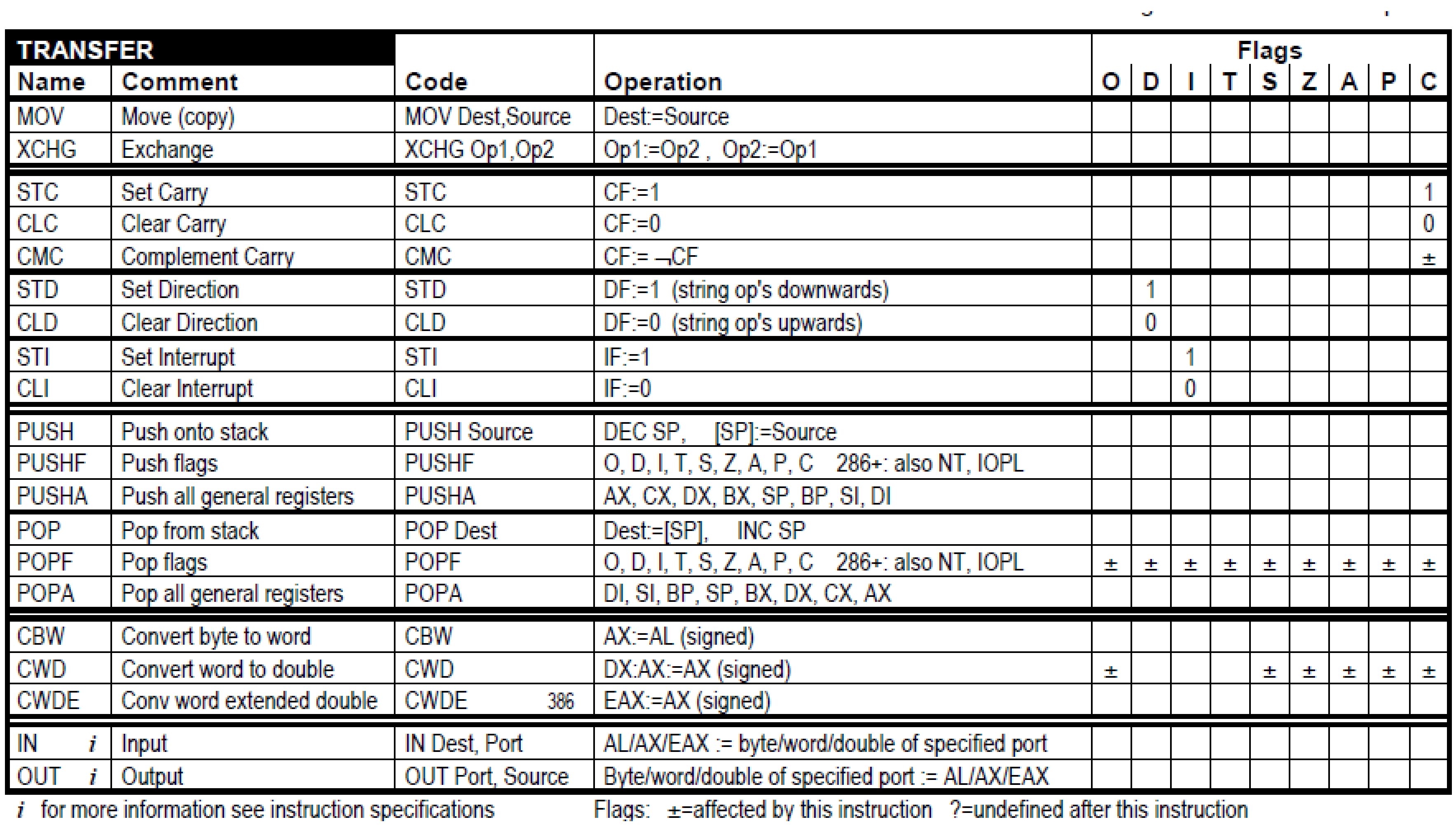
NEG Examples



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

MOVE

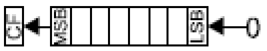
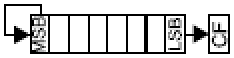
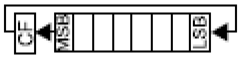
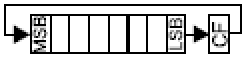
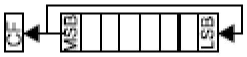
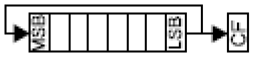


TRANSFER				Flags								
Name	**Comment**	**Code**	**Operation**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC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

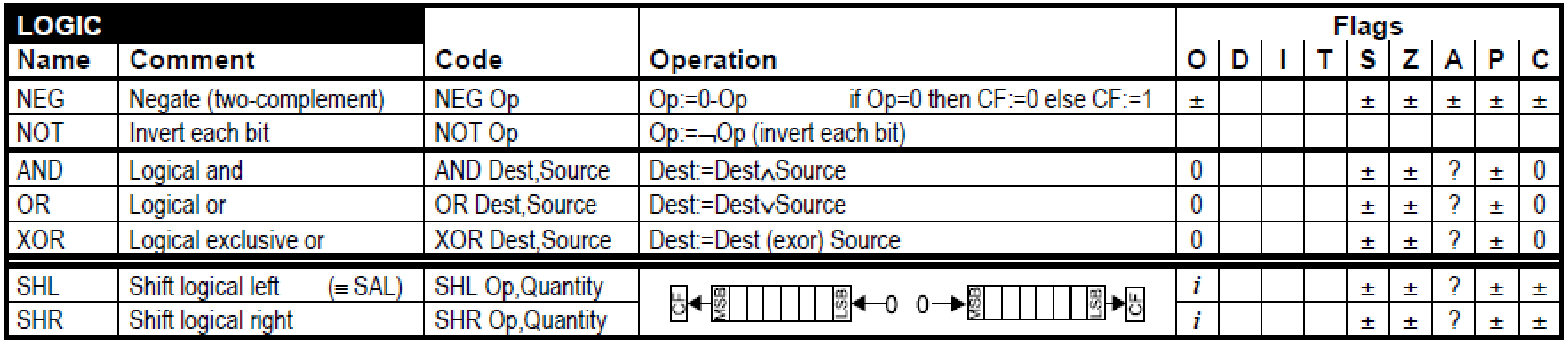
Flags: ±=affected by this instruction ?=undefined after this instruction

Arithmetic

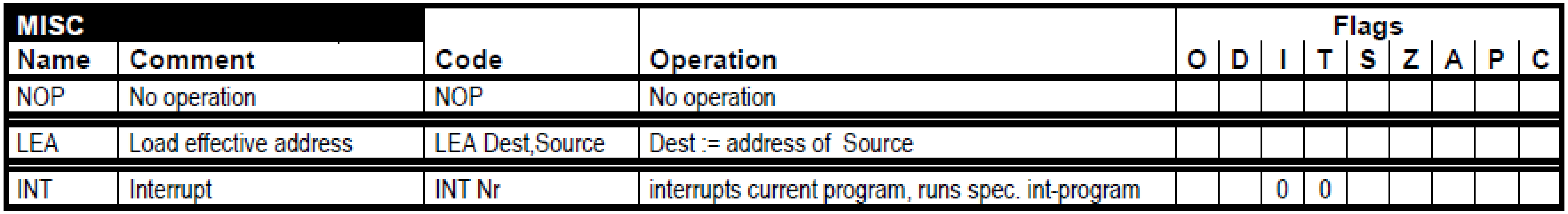
ARITHMETIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	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 +

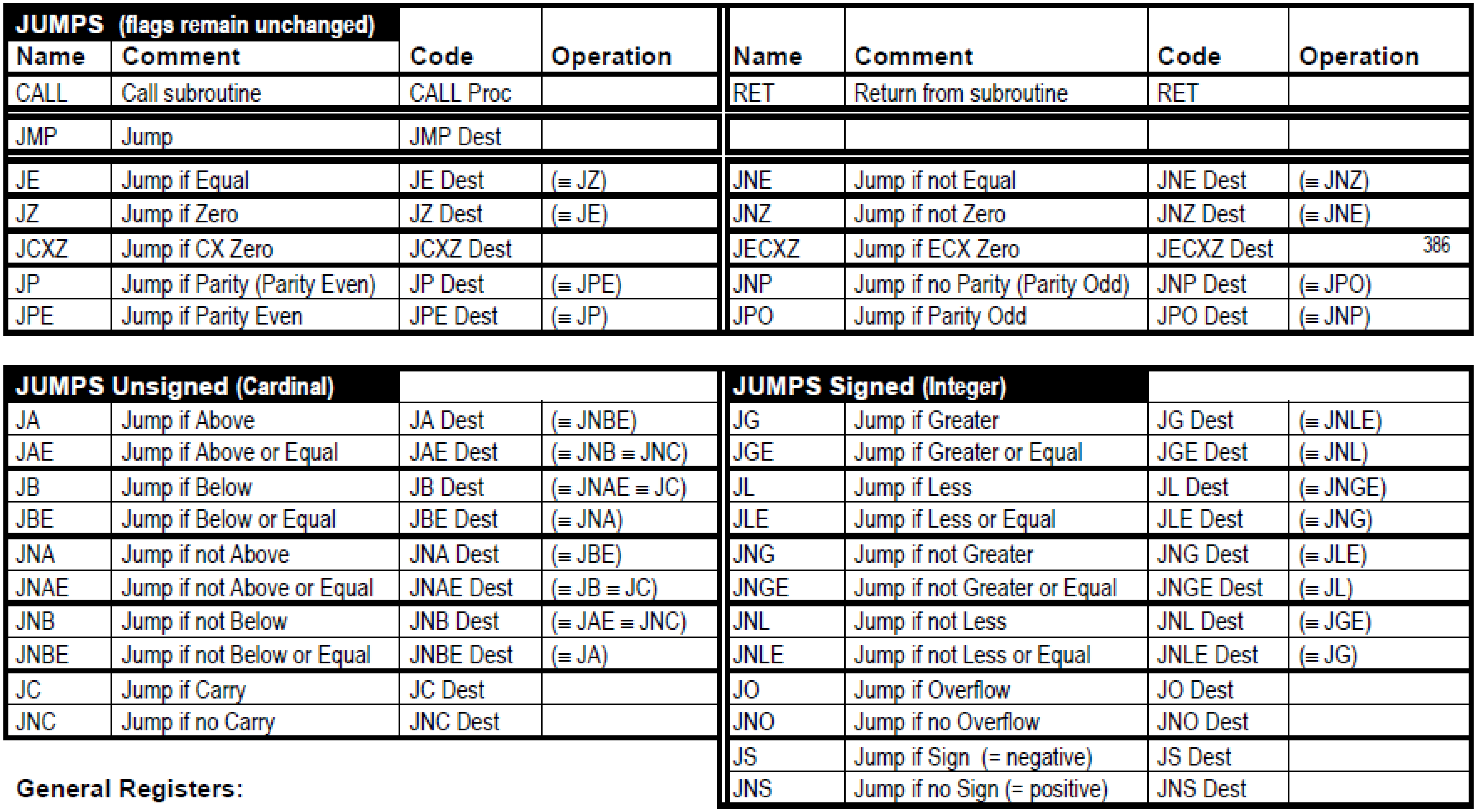


LOGIC				**Flags**								
Name	**Comment**	**Code**	**Operation**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=¬Op (invert each bit)									
AND	Logical and	AND Dest,Source	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	CF ← MSB … LSB ← 0	*i*				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity	0 → MSB … LSB → CF	*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



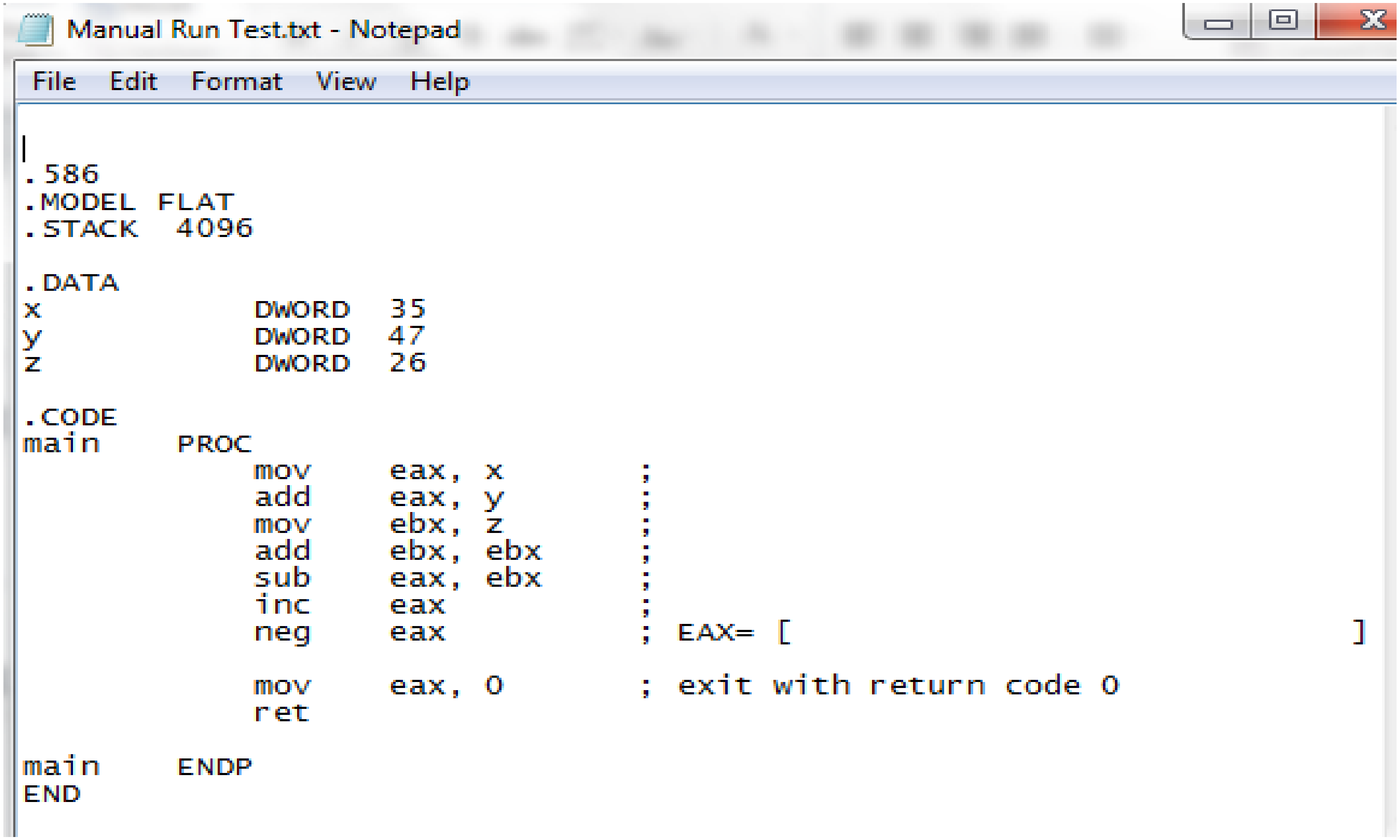
JUMPS (flags remain unchanged)							
Name	Comment	Code	Operation	Name	Comment	Code	Operation
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest	(≡ JZ)	JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)	JNZ	Jump if not Zero	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:

Manual execution practice

- Contents and Flags (CF, ZF, SF, and OF)
- Initially, CF=ZF=SF=OF=0
- Initially, EAX=EBX=00000000



```
.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
```