

# CS271 IA-32 Instructions Reference

ADD – Integer Addition .....	1	OR – Bitwise (Logical) <i>OR</i> .....	32
AND – Bitwise (Logical) <i>AND</i> .....	3	REP – Repeat String Operator .....	33
CALL – Call a Procedure .....	4	REPE, REPZ – Repeat While (Equal, Zero) .....	34
CMP – Compare Operands .....	5	REPNE, REPNZ – Repeat While Not (Equal, Zero) .....	35
CMPSB, CMPSW, CMPSD - Compare String .....	7	RET - Return from Procedure .....	36
DEC – Decrement Stored Value .....	8	SCASB, SCASW, SCASD - Scan String .....	37
DIV – Unsigned Integer Divide .....	9	STD - Set Direction Flag .....	39
IDIV – Signed Integer Divide .....	10	STOSB, STOSW, STOSD - Store String Data .....	40
IMUL – Signed Integer Multiply .....	11	SUB – Integer Subtraction .....	41
INC – Increment Stored Value .....	13	XCHG – Exchange Values .....	43
Jcond – Conditional Jump .....	14	XOR – Bitwise (Logical) <i>Exclusive OR</i> .....	44
JCXZ, JECXZ – Jump if (CX, ECX) = 0 .....	16	<b>FLOATING POINT INSTRUCTIONS... 45</b>	
LODSB, LODSW, LODSD - Load Accumulator from String ....	17	FINIT – Initialize FPU .....	45
LOOP – Loop according to ECX counter .....	18	FLD – Load Floating Point value .....	46
MOV – Copy Stored Value .....	20	FILD – Convert Integer to Float and Load Value .....	47
MOVSb, MOVSW, MOVSD - Copy String .....	21	FST – Store Floating Point Value .....	48
MOVSX – Copy with Sign-extend .....	22	FSTP – Store Floating Point Value and Pop .....	49
MOVZX – Copy with Zero-extend .....	23	FADD – FPU Addition .....	50
MUL – Unsigned Integer Multiply .....	24	FSUB – FPU Subtraction .....	51
NEG – Two’s Complement Negation .....	25	FMUL – FPU Multiplication .....	52
NOT – Bitwise (Logical) <i>NOT</i> .....	25	FDIV – FPU Division .....	53
POP - Pop from Runtime Stack .....	26	FCHS – Change Sign .....	54
POPAD - Pop All GP Registers .....	27	FABS – Absolute Value .....	54
POPFD - Pop EFLAGS Register .....	28		
PUSH - Push on Runtime Stack .....	29		
PUSHAD - Push All GP Registers .....	30		
PUSHFD - Push EFLAGS Register .....	31		



## ADD - Integer Addition

Adds the source operand to the destination operand; can be used to add either signed and unsigned integers. Only one operand can be a memory operand, and both operands must be the same size. The source operand is unchanged by the operation. The result of the operation is stored in the destination operand.

### Syntax

**ADD** destination, source

### Instruction formats

**ADD** reg, reg  
**ADD** reg, imm  
**ADD** mem, reg  
**ADD** mem, imm  
**ADD** reg, mem

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	*

\* changes, (blank) unchanged

#### ; Usage examples 1

```
.data
dwordVal  DWORD 00001200h
.code
MOV  EAX, 23h
MOV  EBX, 0ABCD0000h
ADD  EAX, EBX      ; EAX = ABCD.0023h
ADD  EAX, 11h      ; EAX = ABCD.0034h
ADD  dwordVal, EAX ; dwordVal = ABCD.1234h
ADD  dwordVal, 11h ; dwordVal = ABCD.1245h
MOV  EBX, 1
ADD  EBX, dwordVal ; EAX = ABCD.1246h
```

#### ; Usage examples 2

```
; Example 1      ADD reg, reg
MOV  EAX, 5      ; EAX=5
MOV  EBX, 4      ; EBX=4
ADD  EAX, EBX    ; EAX=9, EBX=4
; Example 2      ADD reg, mem
MOV  EAX, 5      ; EAX=5
ADD  EAX, value_one ; EAX=6
; Example 3      ADD reg, imm
MOV  EAX, 5      ; EAX=5
ADD  EAX, 20     ; EAX=5
```

#### ; Unsigned operations. Carry flag.

; The Carry flag is a copy of the carry out of the most significant bit of the destination operand.

```
XOR  EAX, EAX    ; EAX=0000.0000h
MOV  AL, 0FFh    ; AL = FFh = 255
ADD  AL, 1       ; AL = 00h = 0, CF = 1
```

	1	1	1	1	1	1	1	1	255
+	0	0	0	0	0	0	0	1	1
CF	1	0	0	0	0	0	0	0	

#### ; Unsigned operations. Carry flag.

; No carry out of the highest bit of the destination operand

```
XOR  EBX,EBX    ; EBX = 0000.0000h
MOV  BX,00FFh   ; BX = 00FFh = 255
ADD  BX,1       ; BX = 0100h = 256, CF=0
```

; A carry out of the highest bit of the destination operand

```
XOR  EDX,EDX    ; EDX = 0000.0000h
MOV  DX,0FFFFh  ; DX = FFFFh = 65535
ADD  DX,1       ; DX = 0000h = 0
```

```
; Signed operations.  
; The overflow flag is set when the result  
overflows the destination operand.
```

```
; 8-bit signed value
```

```
XOR    EAX, EAX  
MOV    AL, +127    ; AL=7Fh  
ADD    AL, 1       ; AL=80h=-1, OF=1
```

```
; 16-bit signed value
```

```
XOR    EAX, EAX  
MOV    AX, +32767  ; AL=7FFFh  
ADD    AX, 1       ; AL=8000h=-1, OF=1
```

```
; 32-bit signed value
```

```
XOR    EAX, EAX  
MOV    EAX, +2147483647  
ADD    EAX, 1      ; OF=1
```

```
; PTR operator allows one to operate on data  
in sizes different from their declared type.
```

```
.data
```

```
byteVal BYTE 0FFh
```

```
.code
```

```
MOV    EAX, 0FFFFFFF00h  
ADD    EAX, DWORD PTR byteVal  
                                ; EAX = FFFF.FFFFh  
MOV    BX, 0FF00h  
ADD    BX, WORD PTR byteVal    ; BX=FFFFh
```

## AND - Bitwise (Logical) AND

Performs a bitwise (boolean) AND operation on each pair of the matching bits in the source operand and destination operand. Places the result in the destination operand. For each matching bit, if both corresponding bits in the operands have the value 1, the instruction sets the result to 1; otherwise, it sets the result to 0. The operands must be the same size.

### Syntax

**AND** destination, source

### Instruction format

**AND** reg, reg  
**AND** reg, mem  
**AND** reg, imm  
**AND** mem, reg  
**AND** mem, imm

; AND clears bits without affecting other bits. This technique is called bit masking.

; clears bits 0 and 1

**MOV** AL, 11100111b  
**AND** AL, 11111100b ; AL = 1110.0100b

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	?	PF/PE	*	CF/CY	0

\* changes; 1 sets; 0 clears; ? may change;  
(blank) unchanged

; AND always clears the Overflow and Carry flags. The Sign, Zero, and Parity flags are set or clear according to the result placed in the destination operand

**MOV** AL, 10101010b  
**AND** AL, 01010101b ;AL=0, ZR=0CY=0, OV=0

## CALL - Call a Procedure

Saves on the stack the memory location of the instruction that follows the CALL instruction, then branches to the called procedure's memory location. Later, the RET (return from procedure) instruction brings the program execution back to the memory location saved on the stack. A call is similar to conditional and unconditional jumps; however, the CALL instruction remembers the memory location and can return to this location.

The CALL instruction:

- (1) pushes the offset of the next instruction on the stack (return address);
- (2) loads the offset of the called procedure into the EIP register (called procedure address).

The RET instruction:

- (1) loads the offset on top of the stack (return address) into the EIP register.

Sometimes it is simpler to think of a procedure in assembly language as a function in a high-level language.

### Instruction format

**CALL** label\*      **CALL** register  
**CALL** mem16      **CALL** mem32

\*Please refer to Intel® 64 and IA-32 Architectures Software Developer Manuals for a more detailed look into the distinction between a call to a procedure in the current code segment (near call) and a call to a procedure located in a different segment than the current code segment (far call).

### EFLAGS Register Unchanged

.code

```
main PROC
    CALL sumTwoNums
    XOR EAX, EAX
    exit
main ENDP

sumTwoNums PROC
    MOV EAX, 10
    MOV EBX, 1
    ADD EAX, EBX
    RET
sumTwoNums ENDP

END main
```

offset	main		Execution Step
00401020	call	sumTwoNums	1
00401025	xor	eax, eax	6
00401027	...		
00401029	...		

offset	sumTwoNums		Execution Step
0040102E	mov	eax, 0Ah	2
00401033	mov	ebx, 1	3
00401038	add	eax, ebx	4
0040103A	ret		5

## CMP – Compare Operands

Compares the destination operand to source operand by performing implied subtraction of the source from the destination (destination - source); then sets the status flags in the EFLAGS register in the same manner as does the SUB instruction. Neither operand is modified, and the result is not stored anywhere. Instead, the SUB instruction can be used to keep the result. When a source operand is an immediate value, it is sign-extended to the destination operand's length. CMP is specifically designed to test for conditional jumps.

For comparison of unsigned integers, the Zero Flag and the Carry Flag are crucial. For comparison of signed integers, the Zero Flag, the Overflow Flag, and the Sign flag are crucial.

### Syntax

**CMP** destination, source

### Instruction format

**CMP** reg, reg

**CMP** reg, mem

**CMP** reg, imm

**CMP** mem, reg

**CMP** mem, imm

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI	*	SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	*

\* changes, (blank) unchanged

```
; unsigned (1) (destination < source)
; Zero Flag = 0, Carry Flag = 1.
; Subtracting 2 from 1 requires a borrow.
```

```
MOV AL,1
MOV BL,2
CMP AL,BL          ; ZF/ZR=0, CF/CY=1
```

```
; unsigned (2) (destination > source)
; Zero Flag = 0, Carry Flag = 0
```

```
.data
dwordVal DWORD 1

.code
MOV EAX,2
CMP EAX, dwordVal ; ZF/ZR=0, CF/CY=0
```

```
; unsigned (3) (destination = source)
; Zero Flag = 1, Carry Flag = 0
```

```
MOV AX,1
CMP AX,1          ; ZF/ZR=1, CF/CY=0
```

```
; signed (1) (destination < source)
; Sign Flag != Overflow Flag
```

```
.data
byteVal BYTE -2
.code
MOV AL,+1
CMP byteVal,AL    ; SF/PL=1, OF/OV=0
```

```
; signed (2) (destination > source)
; Sign Flag = Overflow Flag
```

```
.data
wordVal WORD -1
.code
MOV AX, -2
CMP wordVal, AX   ; SF/PL=0, OF/OV=0
```

```
; signed (3) (destination = source)
; Zero Flag = 1
```

```
MOV EAX, -1001
MOV EBX, -1001
CMP EAX, EBX      ; ZF/ZR = 1
```

```

; usage example 1 (character code)

.data
prompt1 BYTE "Yes", 0
.code
XOR EAX,EAX
MOV AL,59h      ; 59h=0111.1001b='Y'
CMP AL,'Y'
je yes
JMP continue
yes:
MOV EDX, OFFSET prompt1
CALL WriteString ; Outputs 'Yes'
continue:
; ...

; Note that ASCII character codes are
stored as integer values.

```

```

; usage example 2 (character code)

.data
prompt1 BYTE "Yes", 0
.code
XOR EAX,EAX
MOV AL,79h      ; 79h=0111.1001b='y'
CMP AL,'Y'
je yes
JMP continue
yes:
MOV EDX, OFFSET prompt1
CALL WriteString
continue:
; Convert to uppercase; clear bit 5
AND AL,11011111b
CALL WriteChar ; Outputs 'Y'
; ...

```

## CMPSB, CMPSW, CMPSD - Compare String

Compares the value stored in one memory location (pointed to by ESI) to the value stored in another memory location (pointed to by EDI). If the *direction flag* is clear, both ESI and EDI are incremented. If the *direction flag* is set, both ESI and EDI are decremented. Like CMP, these set the status flags in the EFLAGS register according to the temporary comparison results. These instructions are compatible with the `REP` prefixes.

Instruction	Operand size	Source op1	Source op2	ESI register	EDI register
CMPSB	BYTE	[ESI]	[EDI]	ESI = ESI ± 1	EDI = EDI ± 1
CMPSW	WORD	[ESI]	[EDI]	ESI = ESI ± 2	EDI = EDI ± 2
CMPD	DWORD	[ESI]	[EDI]	ESI = ESI ± 4	EDI = EDI ± 4

### Instruction format

CMPSB  
CMPSW  
CMPSD

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI	*	SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	*

\* changed, (blank) unchanged

; Example 1

```
.data
value1 WORD 0ABCEh
value2 WORD 0ABCDh

.code
MOV esi, OFFSET value1
MOV edi, OFFSET value2
CMPSW          ; compares words

ja greater     ; ABCEh > ABCDh
; source operand 1 > source operand 2
JMP continue

greater:
MOV  eax, 1    ; outputs 1
CALL WriteDec

continue:
```

; Example 2

```
stringLen EQU LENGTHOF source1

.data
source1 BYTE "THE FIVE BOXING", 0
source2 BYTE "THE FIVE BOXING", 0

.code
CLD          ; move from left to right
MOV  ecx, stringLen
MOV  esi, OFFSET source1
MOV  edi, OFFSET source2
REPE cmpsb
JCXZ allmatch ; ECX=0 if all characters in
               ; source1 and source2 match
JMP  nonmatch

allmatch:
MOV  eax, 1
CALL WriteDec
JMP  continue

nonmatch:
MOV  eax, 0
CALL WriteDec ; Displays 0 (W != L)

continue:
```



## DEC – Decrement Stored Value

Subtracts 1 from a register or memory operand. Does not affect the Carry Flag.

### Operation

Destination  $\leftarrow$  destination – 1

### Instruction format

DEC reg/mem

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	

\* changes, (blank) unchanged

```
MOV AL, 11h ; AL = 11h = 17
DEC AL      ; AL = 10h = 16
```

; The instruction treats integers as unsigned values.

```
MOV AX, 1 ; AX = 0001h = 1
DEC AX    ; AX = 0000h = 0
DEC AX    ; AX = FFFFh = 255
DEC AX    ; AX = FFFEh = 254
```

; DEC instruction does not update the Carry Flag

```
MOV AL, 0
DEC AL ; Carry Flag = 0
```

; To update the Carry Flag, use a SUB instruction with an immediate source operand of 1.

```
MOV AL, 0
SUB AL, 1 ; Carry Flag = 1
```

## DIV - Unsigned Integer Divide

Carries out 8-bit, 16-bit, or 32-bit unsigned integer division. Returns a quotient and a remainder.

*Dividend* -- the integer to be divided. *Divisor* -- the integer to divide by. *Quotient* -- the result.

The syntax "EDX:EAX" indicates that the 4 bytes of EDX and the 4 bytes of EAX are seen as a single 8-byte value, with EDX holding the most significant 4 bytes and EAX holding the least significant 4 bytes of this pseudo-8-byte register. The dividend is overwritten.

Dividend	Dividend Size	Divisor	Divisor Size	Quotient	Remainder	Before DIV*
AX	16 bits	reg8/mem8	8 bits	AL	AH	<code>MOV AH, 0</code>
DX:AX	32 bits	reg16/mem16	16 bits	AX	DX	<code>MOV DX, 0</code>
EDX:EAX	64 bits	reg32/mem32	32 bits	EAX	EDX	<code>MOV EDX, 0</code>

\* For 8-bit DIV, AH must be set or cleared. To clear use `MOV AH, 0` or `XOR AH, AH`.

\* For 16-bit DIV, DX must be set or cleared. To clear use `MOV DX, 0` or `XOR DX, DX`.

\* For 32-bit DIV, EDX must be set or cleared. To clear use `MOV EDX, 0` or `XOR EDX, EDX`.

### Instruction format

```
DIV reg8      DIV mem8
DIV reg16     DIV mem16
DIV reg32     DIV mem32
```

### EFLAGS changed

OF/OV	?	DF/UP		IF/EI		SF/PL	?
ZF/ZR	?	AF/AC	?	PF/PE	?	CF/CY	?

?: may change, (blank): unchanged

`DIV reg8` (divide AX by BL)

```
MOV AX, 00AFh ; dividend AX = 00AFh
MOV BL, 2     ; divisor BL = 02h
DIV BL        ; quotient AL = 57h
              ; remainder AH = 01h
```

`DIV mem32` (divide EDX:EAX by variable)

```
.data
divisor  DWORD 00000100h
.code
MOV EDX, 0Ah      ; EDX = 0000000Ah
MOV EAX, 0B0C0D0Eh ; EAX = 0B0C0D0Eh
DIV divisor        ; EAX = 0A0B0C0Dh
                  ; EDX = 0000000Eh
```

`div reg16` (divide DX:AX by BX)

```
MOV AX, 0AAAAh ; low dividend AX=AAAAh
MOV DX, 0       ; clear high dividend
MOV BX, 1000h   ; BX = 1000h
DIV BX          ; AX = 000Ah (10d)
                ; DX = 0AAAh (2730d) remainder
```

`DIV reg32` (divide EDX:EAX by EBX)

```
MOV EAX, 500     ; EAX = 000001F4h
MOV EDX, 0       ; clear high dividend
MOV EBX, 21      ; EBX = 00000015h
DIV EBX          ; EAX = 23
                  ; EDX = 17 (remainder)
```

## IDIV – Signed Integer Divide

Carries out 8-bit, 16-bit, and 32-bit signed integer division. Returns a quotient and a remainder. The remainder always has the same sign as the dividend.

*Dividend*: the integer to be divided. *Divisor*: the integer to divide by. *Quotient* -- the result.

Dividend	Dividend Size	Divisor	Divisor Size	Quotient	Remainder	Before IDIV*
AX	16 bits	reg8/mem8	8 bits	AL	AH	CBW
DX:AX	32 bits	reg16/mem16	16 bits	AX	DX	CWD
EDX:EAX	64 bits	reg32/mem32	32 bits	EAX	EDX	CDQ

\* For 8-bit IDIV, AL must be sign-extended into AH: CBW (convert BYTE to WORD).

\* For 16-bit IDIV, AX must be sign-extended into DX: CWD (convert WORD to DWORD).

\* For 32-bit IDIV, EAX must be sign-extended into EDX: CDQ (convert DWORD to QWORD).

### Instruction format

```
IDIV reg8      IDIV mem8
IDIV reg16     IDIV mem16
IDIV reg32     IDIV mem32
```

### EFLAGS changed

OF/OV	?	DF/UP		IF/EI		SF/PL	?
ZF/ZR	?	AF/AC	?	PF/PE	?	CF/CY	?

? may change, (blank) unchanged

**IDIV reg8** (divide **AX** by **BL**)

```
MOV AL,-128 ; dividend AX=??80h
CBW         ; AX=FF80h
MOV BL,+3   ; divisor  BL=03h
IDIV BL     ; quotient AL=D6h= -42
             ; remainder AH=FEh= -2
```

**IDIV mem32** (divide **EDX:EAX** by divisor)

```
.data
divisor SDWORD -401 ; FFFFFFFE6Fh
.code
MOV EAX, -271400 ; EAX=FFFBDBD8h, EDX=?
CDQ             ; EDX=FFFFFFFFh

IDIV divisor    ; EAX=-676,EDX = -324
```

**IDIV reg16** (divide **DX:AX** by **BX**)

```
MOV AX,-8086 ; dividend AX=E06Ah
             ; DX=?
CWD          ; DX=FFFF
MOV BX,+512  ; divisor  BX=0200h

IDIV BX      ; quotient AX=FFF1h= -15
             ; remainder DX=FE6Ah= -406
```

## IMUL - Signed Integer Multiply

Multiplies signed integers. Has three formats. Sign-extends the highest bit of the lower half of the product into the upper half of the product. Can also multiply unsigned integers -- the result must not use the most significant bit of the destination.

### IMUL multiplier

Behaves as the MUL instruction. The multiplicand and the destination are implied. The multiplier is multiplied with the implied multiplicand; the product is placed in the destination.

Multiplicand (in the accumulator register)	Multiplicand Size (bits)	Multiplier	Multiplier Size (bits)	Product Returned In (overwrites the contents of)	Product Size (bits)
AL	8 bits	reg8/mem8	8 bits	AX	16 bits
AX	16 bits	reg16/mem16	16 bits	DX:AX	32 bits
EAX	32 bits	reg32/mem32	32 bits	EDX:EAX	64 bits

### IMUL destination, multiplier

The destination is the multiplicand and the source is the multiplier. The product is truncated to the length of the destination.

### IMUL destination, multiplicand, multiplier

The destination, multiplicand, and multiplier are specified. The product is truncated to the length of the destination.

### Instruction format

#### ; One operand

```
IMUL reg8      IMUL reg16      IMUL reg32
IMUL mem8      IMUL mem16      IMUL mem32
```

#### ; Two operands

```
IMUL reg16, reg/mem16
IMUL reg16, imm8      IMUL reg16, imm16
```

```
IMUL reg32, reg/mem32
IMUL reg32, imm8      IMUL reg32, imm32
```

#### ; Three operands

```
IMUL reg16, reg/mem16, imm8
IMUL reg16, reg/mem16, imm16
IMUL reg32, reg/mem32, imm8
IMUL reg32, reg/mem32, imm32
```

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	?
ZF/ZR	?	AF/AC	?	PF/PE	?	CF/CY	*

\* changes, (blank) unchanged

#### One operand

Carry and Overflow flags are set if the upper half of the product is not a sign extension of the lower half.

#### Two or three operands

The product is truncated to the length of the destination. Overflow and Carry flags are set if significant digits of the product are lost.

<pre> ; imul reg8  MOV AL,-10;multiplicand AL=-10=F6h MOV BL,10 ;multiplier  BL= 10=0Ah IMUL BL ;product      AX=-100=FF9Ch  ; AH is the sign extension of AL. ; Both Carry and Overflow flags are clear. </pre>	<pre> ; mul reg8  MOV AL,-10;multiplicand AL=-10=F6h MOV BL,10 ;multiplier  BL= 10=0Ah MUL BL ;product      AX=2460=099Ch  ; The MUL instruction does not preserve the sign of the product. ; AH is not the sign extension of AL. </pre>
<pre> ; imul reg16, imm8  MOV BX, -4000h ; BX = -16384 IMUL BX, 2 ; BX = -32768  ; -16,384 * 2 = -32,768 = 8000h  ; Since -2^15 = -32,768, the product fits in the 16-bit register. The product is not truncated.  ; Overflow and carry flags are clear. </pre>	<pre> ; imul reg16, imm8  MOV BX, -4000h ; BX = -16384 IMUL BX, 10 ; BX = -32768  ; -16384 * 10 = -163,840 = FFFD 8000h  ; The destination operand has 16 bits, so the FFFD is truncated.  ; Overflow and carry flags are set. </pre>
<pre> ; imul reg32, mem32, imm8  .data multiplicand DWORD -1073741824 .code IMUL EAX, multiplicand, 2  ; EAX = 8000 0000h = -2,147,483,648.  ; Overflow and carry flags are clear. </pre>	<pre> ; imul reg32, mem32, imm8  .data multiplicand DWORD -1073741824 .code IMUL EAX, multiplicand, 3  ; EAX = 40000000h = 1,073,741,824.  ; -1073741824 * 3 = -3,221,225,472, ; or ; -C000.0000h*3h=FFFF.FFFF.4000.0000h  ; The maximal signed integer that fits ; in a 32-bit register is ; 8000.0000h, or -2,147,483,648.  ; Since the significant digits are lost, the overflow and carry flags are set. </pre>

## INC – Increment Stored Value

Adds 1 to a register or memory operand. Does not affect the Carry Flag.

### Instruction Format

**INC** reg/mem

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	

\* changes, (blank) unchanged

```
MOV AL, 10h ; AL = 10h = 16
INC AL      ; AL = 11h = 17
```

```
; Example 1      INC reg
MOV EAX, 5        ; EAX = 5
INC EAX           ; EAX = 6

; Example 2      INC mem
MOV someDword, 5  ; someDword = 5
INC someDword     ; someDword = 6
```

```
.data
wordVal WORD 0FFFh

.code

INC wordVal ; wordVal = 1000h
```

```
; The instruction treats integers as
unsigned values.

MOV AX, 0FFFFh ; AX = FFFFh = 255
INC AX         ; AX = 0000h = 0, ZF=1,
INC AX         ; AX = 0001h = 1, ZF=0
```

```
; INC does not update the CF flag

MOV AL, 0FFh
INC AL ; Carry Flag = 0
```

```
; To update the Carry Flag, use a ADD
instruction with an immediate source
operand of 1.

MOV AL, 0FFh
ADD AL, 1 ; Carry Flag = 1
```

## Jcond – Conditional Jump

Jumps to a destination code label if a specific condition is met. Conditions are evaluated based on the flags in the EFLAGS register. If the condition is not met, the program executes the instruction *immediately following* the Jcond instruction.

Any arithmetic, comparison, or boolean instruction (CMP and TEST being the most typical) that precedes a conditional jump will set flag values in the Status Register and allow the Jcond to be used.

Signed and unsigned values require different conditional jump instructions. Conditional jumps cannot refer to a destination label that is more than 32-bits away in the code segment.

### Instruction Format

jcondition destination\_Label

### EFLAGS Register Unchanged

### Jumps Named for Operand Comparisons

#### ...for Signed Comparisons

Jcond	Condition <i>leftOp __ rightOp</i>	Flags
JE	equal to	ZF = 1
JNE	not equal to	ZF = 0
JG	greater than	SF = 0 and ZF = 0
JGE	greater than or equal to	SF = OF
JNG	not greater than	ZF = 1 or SF ≠ OF
JNGE	not greater than or equal to	SF ≠ OF
JL	less than	SF ≠ OF
JLE	less than or equal to	ZF = 1 or SF ≠ OF
JNL	not less than	SF = OF
JNLE	not less than or equal to	SF = 0 and ZF = 0

#### ...for Unsigned Comparisons

Jcond	Condition <i>leftOp __ rightOp</i>	Flags
JE	equal to	ZF = 1
JNE	not equal to	ZF = 0
JA	above	CF = 0 and ZF = 0
JAЕ	above or equal to	CF = 0
JNA	not above	CF = 1 or ZF = 1
JNAЕ	not above or equal to	CF = 1
JB	below	CF = 1
JBE	below or equal to	CF = 1 or ZF = 1
JNB	not below	CF = 0
JNBE	not below or equal to	CF = 0 and ZF = 0

### Jumps Named for Flag Status

Jcond	Condition	Flags	Same as
JO	signed overflow	OF = 1	-
JNO	no signed overflow	OF = 0	-
JC	unsigned carry	CF = 1	JB
JNC	no unsigned carry	CF = 0	JA
JZ	zero result	ZF = 1	JE
JNZ	non-zero result	ZF = 0	JNE
JS	negative result	SF = 1	-
JNS	positive result	SF = 0	-
JP	parity in result	PF = 1	JPE
JNP	no parity in result	PF = 0	JPO
JPE	result even parity	PF = 1	JP
JPO	result odd parity	PF = 0	JNP

```
; CMP tests a condition.
; Jump if equal.
```

```
MOV EBX, 0FFFFFFFh
ADD EBX, 2
CMP EBX, 1 ; EBX=1
JE is_one ; Zero Flag=1
```

```
not_one:
MOV EAX, -1
CALL WriteDec
JMP continue
```

```
is_one:
MOV EAX, 1
CALL WriteDec
```

```
continue:
CALL CrLf
```

```
; SUB tests a condition.
; Jump if equal
```

```
MOV EBX, 0FFFFFFFh
ADD EBX, 2
SUB EBX, 1 ; EBX=0
JE is_one ; Zero Flag=1
```

```
not_one:
MOV EAX, -1
CALL WriteDec
JMP continue
```

```
is_one:
MOV EAX, 1
CALL WriteDec
```

```
continue:
CALL CrLf
```

```
; CMP tests a condition.
; Jump if zero.
```

```
MOV EBX, 0FFFFFFFh
ADD EBX, 2
CMP EBX, 1 ; EBX=1
JZ is_one ; Zero Flag=1
```

```
not_one:
MOV EAX, -1
CALL writedec
JMP continue
```

```
is_one:
MOV EAX, 1
CALL writedec
```

```
continue:
CALL CrLf
```

```
; SUB tests a condition.
; Jump if zero.
```

```
MOV EBX, 0FFFFFFFh
ADD EBX, 2
SUB EBX, 1 ; EBX=0
JZ is_one ; Zero Flag=1
```

```
not_one:
MOV EAX, -1
CALL WriteDec
JMP continue
```

```
is_one:
MOV EAX, 1
CALL WriteDec
```

```
continue:
CALL CrLf
```



## JCXZ , JECXZ - Jump if (CX, ECX) = 0

JCXZ checks the CX register's value and then jumps to a destination label if the CX's value equals zero. If the value is not zero, the program executes an instruction that immediately follows the conditional jump instruction.

Likewise, JECXZ checks whether the ECX register's value equals zero; the program then branches according to this evaluation result.

The destination label must be within +128 to +127 bytes of the instruction; the signed 8-bit offset value is added to the instruction pointer.

### Instruction format

```
jcxz destinationLabel
```

```
jecz destinationLabel
```

### EFLAGS Register Unchanged

```
; The program writes 0 to the console  
window
```

```
MOV cx, 0FFFFh  
INC cx  
JCXZ outputResult ; jumps to label  
MOV EAX, 1  
CALL WriteDec  
JMP continue  
outputResult:  
XOR EAX, EAX  
MOV AX, cx  
CALL WriteDec ; 0  
continue:
```

```
; The program writes 0 to the console  
window
```

```
XOR ECX, ECX  
JECXZ outputResult ; jumps to label  
MOV EAX, 1  
CALL WriteDec  
JMP continue  
outputResult:  
MOV EAX, ECX  
CALL WriteDec ; 0  
continue:
```

## LODSB, LODSW, LODSD - Load Accumulator from String

Loads a value from memory location (string or array), pointed to by ESI, into the accumulation register. If the direction flag is clear, ESI is incremented. If the direction flag is set, ESI is decremented. These instructions are compatible with the `REP` prefixes. Often, a `LOOP` construct encompasses these instructions and processes data loaded into the register.

Instruction	Value size	Loads into	ESI Change	Equivalent Instructions (DF=0)	Equivalent Instructions (DF=1)
LODSB	BYTE	AL	ESI = ESI ± 1	mov al, [esi] inc esi	mov al, [esi] dec esi
LODSW	WORD	AX	ESI = ESI ± 2	mov ax, [esi] add esi, 2	mov ax, [esi] sub esi, 2
LODSD	DWORD	EAX	ESI = ESI ± 4	mov eax, [esi] add esi, 4	mov eax, [esi] sub esi, 4

### Instruction format

LODSB  
LODSW  
LODSD

### EFLAGS Register Unchanged

```
.data
string BYTE 36, 56, 57, 67, -16, \
57, 67, -16, 19, 35, 2, 7, 1, -15
strLen DWORD LENGTHOF string

.code
CLD          ; clears direction flag
MOV ecx, strLen
MOV esi, OFFSET string

convert:
LODSB        ; increments ESI
ADD al, '0'   ; converts to ASCII
CALL WriteChar
LOOP convert  ; This is CS271!
```

```
.data
string BYTE 36, 56, 57, 67, -16, \
57, 67, -16, 19, 35, 2, 7, 1, -15
strLen DWORD LENGTHOF string

.code
MOV ecx, strLen
MOV esi, OFFSET string

convert:
MOV al, [esi]
INC esi      ; increments ESI
ADD al, '0'
CALL WriteChar
LOOP convert
```

## LOOP - Loop according to ECX counter

Decrements ECX by 1, then checks ECX for 0. If ECX equals 0, the loop terminates, and the program executes the instruction that immediately follows the LOOP instruction. If ECX is not equal to 0, the program execution jumps to the label's destination. Before the loop starts, load the number of iterations into ECX.

At the machine code level, an assembly language label is converted to a signed 8-bit immediate value. Thus, the destination specified by the label must be within the range of [-128, +127] bytes of the current instruction's location. If the jump to the destination specified by the label exceeds the specified range, the assembler will produce an error message.

The LOOP does not affect the EFLAGS. When the ECX value becomes 0, the Zero Flag is not set.

### Instruction format

**LOOP** destination

### EFLAGS Register Unchanged

; Usage example

```
MOV EAX,0
MOV ECX,3 ; set the count
```

destination:

```
INC EAX ; EAX: 1 -> 2 -> 3
LOOP destination ; ECX: 2 -> 1 -> 0
```

; after the loop terminates, EAX = 3

```
MOV EAX, 0 ; EAX = 0
```

; AVOID this common error (!)

```
MOV ECX,1 ; set the count
```

destination:

```
INC ECX ; ECX: 2->2-> ...
LOOP destination ; ECX: 1->1-> ...
```

; This loop never terminates because ECX never reaches 0.

; Use JECXZ instruction to check whether ECX contains 0.

```
MOV EBX, 0
MOV ECX, 0
JECXZ continue ; prevent the error
```

destination:

```
INC EBX
LOOP destination
```

continue:

...

; JECXZ jumps to the destination label 'continue' if ECX = 0.

; AVOID this common error (!)

```
MOV EAX,0
MOV ECX,0 ; DO NOT SET COUNT TO 0
```

destination:

```
INC EAX
LOOP destination
; ECX = 0-1 = 4,294,967,295
```

; The loop will repeat 4,294,967,296 times, increment EAX 4,294,967,296 times, and result in an overflow of the value stored in EAX. After the loop terminates, EAX = 0.

```

; A nested loop

.data
count      DWORD 0
multiplicand  DWORD 2

.code
MOV EAX, 1
MOV ECX, 3 ; the outer loop's count

destination_1:
; save outer loop's count
MOV count, ECX
; set the inner loop's count
MOV ECX, 2

destination_2:
MUL multiplicand
; repeats the inner loop 2 times
LOOP destination_2
; restore outer loop count
MOV ECX, count
; repeats the outer loop 3 times
LOOP destination_1

; For each iteration of the outer loop, the
inner loop iterates 2 times. EAX contains 64,
or, equivalently, 2 raised to the power of
6.

```

## MOV – Copy Stored Value

Copies data from a source operand to a destination operand.

After the MOV instruction has been executed, both the source and destination contain the same value. The destination operand's contents are replaced; the source operand contents are unchanged. The operands must match in size, cannot both be memory operands. The instruction pointer register (EIP) cannot be a destination.

### Instruction format

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

### EFLAGS Register Unchanged

; Immediate value data transfer

```
MOV AX, 271 ; Imm to regter
MOV memory, 8086 ; Imm to mem direct
MOV arr[EBX], 1 ; Imm to mem indirect
```

; Memory to memory data transfer

```
.data
value1 DWORD 0
value2 DWORD 1
.code
MOV EAX, value1 ; value1=0, value2=1
MOV value2, EAX ; value1=0, value2=0
```

; Register data transfer

```
MOV EAX, 271 ; AX=0000010Fh=271, var=0
```

; Register to memory direct:

```
MOV var, EAX ; var = 271
```

; Register to memory indirect:

```
MOV EDX, 0
MOV array[EDX], AL ; arr[0]=0Fh=15
```

; Register to register:

```
XOR EBX, EBX ; EBX = 0
MOV BL, 8 ; BL = 8
MOV EAX, EBX ; EAX = 8
```

; Direct/indirect memory data transfer

```
.data
var DWORD 0
arr BYTE 10, 11
.code
; -- (1) Indirect memory transfer --
; Memory direct to register:
MOV EBX, var ; EBX = 0
; -- (2) Indirect memory transfer --
; Memory indirect to register:
MOV dl, arr[EBX] ; dl = 0Ah = 10
INC BL ; (!)
MOV dl, arr[EBX] ; dl = 0Bh = 11
```

## MOVSb, MOVSW, MOVSD - Copy String

Copies the value from one memory location, pointed to by ESI, into another memory location, pointed to by EDI. If the *direction flag* is clear, both ESI and EDI are incremented. If the *direction flag* is set, both ESI and EDI are decremented. These instructions are compatible with the REP prefixes. Often, a LOOP construct encompasses these instructions and processes data loaded into the register.

Instruction	Value size	Loads from	Loads into	ESI Change	EDI Change
MOVSb	BYTE	[ESI]	[EDI]	ESI = ESI ± 1	EDI = EDI ± 1
MOVSW	WORD	[ESI]	[EDI]	ESI = ESI ± 2	EDI = EDI ± 2
MOVSD	DWORD	[ESI]	[EDI]	ESI = ESI ± 4	EDI = EDI ± 4

### Instruction format

MOVSb  
MOVSW  
MOVSD

### EFLAGS Register Unchanged

; Usage example 1

```
.data
source WORD 2 DUP (1234h) ;1234h,1234h
destination WORD 2 DUP (0) ;0000h,0000h

.code
CLD ; clears direction flag
MOV ecx, LENGTHOF source ; sets count
MOV esi, OFFSET source ; from
MOV edi, OFFSET destination ; to
REP MOVSW ; Copies 2 WORD

; destination:
; 1234h, 1234h, 1234h
```

; Usage example 2

```
.data
source BYTE 10 DUP (0FFh)
destination BYTE 20 DUP (?)

.code
CLD
MOV ecx, LENGTHOF source ; count = 20
MOV esi, OFFSET source
MOV edi, OFFSET destination
REP MOVSB ; Copies 20 BYTES

; destination:
; Fh, Fh, Fh, Fh, Fh, \
; Fh, Fh, Fh, Fh, Fh, \
; Fh, Fh, Fh, Fh, Fh, \
; Fh, Fh, Fh, Fh, Fh, \
```

## MOVSX – Copy with Sign-extend

Copies the signed value from a smaller-sized source operand into a larger-sized destination operand, and sign extends this value into the upper bits of a 16-bit or 32-bit register. This instruction extends and copies a value in one step.

Notice the difference between extending signed integers and unsigned integers -- for instance, zero-padding the high bits changes the negative number's value. MOVSX preserves the sign of the signed integer when it extends the integer to higher bits; it assumes that the value being moved is in the signed integer format.

### Instruction format

```
MOVSX reg32, reg/mem8
MOVSX reg32, reg/mem16
MOVSX reg16, reg/mem8
```

### EFLAGS Register Unchanged

```
.data
byteVal BYTE 10000000b ; 80h = -128
```

```
.code
MOV EAX, 0 ; EAX = FFFFFFFF80h = -128
MOVSX EAX, byteVal
```

; The value -128 (1000 0000b) moved and sign-extended to the EAX register produce -128 (FFFF FF80h) in the EAX register.

```
.data
byteVal BYTE 10000000b ; 80h = -128
```

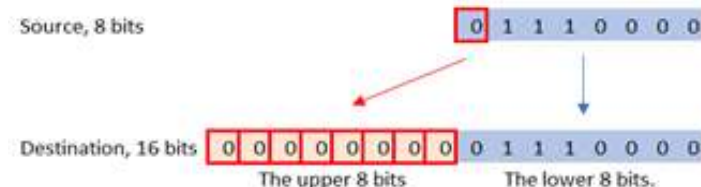
```
.code
MOV EAX, 0 ; EAX = 00000080h = 128
MOV AL, byteVal
```

; The value -128 (1000 0000b) moved to the lowest 8 bits of the EAX register produce 128 (0000 0080h) in the signed integer notation.

```
.data
byteVal BYTE 70h ; 0111 0000b
```

```
.code
MOV EAX, 0
MOVSX AX, byteVal ; AX = 0070h
```

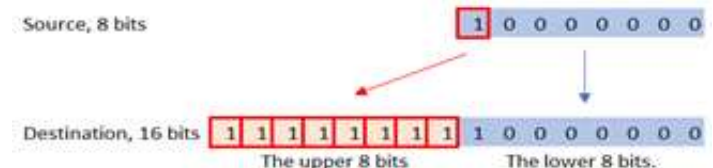
The most significant hexadecimal digit of byteVal is less than or equal to 7. Since the highest bit of the byteVal is not set, movsx copies 0s into the extended bits of the destination operand.



```
.data
byteVal BYTE 80h ; 1000 0000b
```

```
.code
MOV EAX, 0
MOVSX AX, byteVal ; AX = FF80h
```

The most significant hexadecimal digit of byteVal is greater than 7. Since the highest bit of the byteVal is set, movsx copies 1s into the extended bits of the destination operand.



```
MOV BX, 271h
MOVSX EAX, BX ; EAX = 0000 0271h
MOVSX EAX, BL ; EAX = 0000 0071h
```

```
MOV BX, 0A080h
MOVSX EAX, BX ; EAX = FFFF A080h
MOVSX EAX, BL ; EAX = FFFF FF80h
```

## MOVZX – Copy with Zero-extend

Copies the unsigned value from a smaller-sized source operand into a larger-sized destination operand, zero-extending this value into the upper bits of a 16-bit or 32-bit register. Ensures that all of the leading bits are set to zero after converting an unsigned integer value to the destination operand's higher bits. Zero extension and data transfer are executed in one step.

### Instruction format

```
MOVZX reg32, reg/mem8
MOVZX reg32, reg/mem16
MOVZX reg16, reg/mem8
```

### EFLAGS Register Unchanged

```
MOV AX, 8000h ;AX = 8000h

MOV BL, 0ABh ;BL = ABh
MOVZX AX, BL ;AX = 00ABh, EAX=????00ABh
MOVZX EAX, BL ;EAX = 000000ABh
```

```
MOV AX, 8000h ;AX=8000h

MOV BL, 0ABh ;BL=ABh
MOV AL, BL ;AX=80ABh, EAX=????80ABh
```

```
MOV BX, 0A0B1h

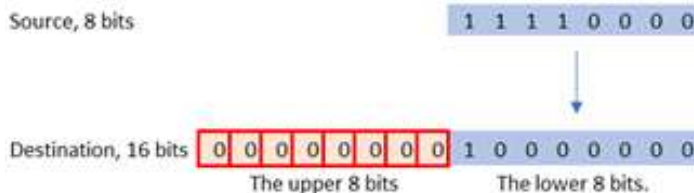
MOVZX EAX, BX ; EAX = 0000.A0B1h
MOVZX ECX, BL ; ECX = 0000.00B1h
MOVZX DX, BL ; DX = 00B1h
```

```
.data
byteVal BYTE 0B1h
wordVal WORD 0A0B1h

.code
MOVZX EAX, wordVal ; EAX = 0000.A0B1h
MOVZX ECX, byteVal ; ECX = 0000.00B1h
MOVZX DX, byteVal ; DX = 00B1h
```

```
.data
byteVal BYTE 11110000b

.code
MOVzx AX, byteVal
; AX = 0000.0000.1111.0000b
```





## MUL – Unsigned Integer Multiply

Multiplies unsigned integers. The destination operand is implied; the instruction line contains one operand. Result must be returned in the register twice the size of the source operand. *Does not preserve the sign of the product.*

Multiplicand (in the accumulator)	Multiplicand Size (bits)	Multiplier	Multiplier Size (bits)	Product Returned In (overwrites the contents of)	Product Size (bits)
AL	8 bits	reg8/mem8	8 bits	AX	16 bits
AX	16bits	reg16/mem16	16 bits	DX:AX	32 bits
EAX	32 bits	reg32/mem32	32 bits	EDX:EAX	64 bits

**NOTE:** Prior to using the 16-bit or 32-bit operand version, save the contents of EDX or DX.

### Instruction format

```
MUL reg8      MUL mem8
MUL reg16     MUL mem16
MUL reg32     MUL mem32
```

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	?
ZF/ZR	?	AF/AC	?	PF/PE	?	CF/CY	*

\* changes, (blank) unchanged

```
; MUL reg8
```

```
MOV AL,0Ah ; multiplicand AL=0Ah
MOV BL,10h ; multiplier  BL=10h
MUL BL     ; product    AX=00 A0h,
           ; where      AH=0, AL=A0h.
```

```
; Because the upper half of the product
equals 0 (AH = 0), the carry flag is clear
(CF = 0).
```

```
; MUL mem32
```

```
.data
multiplicand DWORD 0ABCDEFh
.code
MOV EAX, multiplicand
MOV EBX, 100h
MUL EBX ; EDX:EAX = 00000000:ABCDEF000h
```

```
; Because the upper half of the product
equals zero, the carry flag is clear, CF =
0.
```

```
; MUL reg16
```

```
MOV AX,0A000h ;AX = A000h
MOV BX,10h    ;BX = 10h
MUL BX       ;DX:AX = 000A.0000h, where DX=000Ah.
```

```
; Because the upper half of the product is not equal to zero,
; the carry flag is set (CF = 1).
```

## NEG – Two's Complement Negation

Replaces the value in the destination operand with its two's complement (negation). Equivalent to (and faster than) multiplying by -1. Equivalent to subtracting the operand from 0.

The Carry flag is set to 0 if the operand is 0, otherwise it is set to 1. Overflow, Zero, Sign, Aux Carry, and Parity flags are set according to the result.

### Instruction format

NEG reg  
NEG mem

### EFLAGS changed

OF/OV	?	DF/UP		IF/EI		SF/PL	?
ZF/ZR	?	AF/AC	?	PF/PE	?	CF/CY	?

\* changes, ? may change, (blank) unchanged

; Negate positive signed value in register

```
MOV EAX, 271 ; Imm to register
NEG EAX      ; EAX = -271
```

; Negate negative signed value in memory

```
.data
value1 DWORD -400

.code
NEG value1 ; value1 = 400
```

; Negate 0-value

```
MOV AX, 0
NEG AX ; AX = 0, Carry Flag = 1
```

## NOT – Bitwise (Logical) NOT

Performs a bitwise NOT operation on the destination operand.  
Each 1-bit in the operand is set to 0, each 0-bit is set to 1.

### Instruction format

NOT reg  
NOT mem

### EFLAGS changed

OF/OV	?	DF/UP		IF/EI		SF/PL	?
ZF/ZR	?	AF/AC	?	PF/PE	?	CF/CY	?

\* changes, ? may change, (blank) unchanged

; Invert 8-bit register value

```
MOV AL, 01010000b ; Imm to register
NOT AL            ; AL = 10101111b
```

; Invert 16-bit memory value

```
.data
value1 DWORD 00FFh

.code
NOT value1 ; value1 = 0FF00h
```

## POP - Pop from Runtime Stack

Loads the value from the stack's top (ESP points to this value) to the location specified by the destination operand. Then increments the value of ESP. If the destination operand is 16 bits, POP increments the offset value, which is stored in ESP, by 2. If the destination operand is 32 bits, POP increments the offset value, which is stored in ESP, by 4.

### Instruction format

**POP** reg16/mem16

**POP** reg32/mem32

### EFLAGS Register Unchanged

```
; Please take a look at PUSH
; instruction usage example.
; Here, we push the values onto
; the stack:
```

**.data**

**a**        **WORD**    **0ABCDh**

**b**        **DWORD** **0FFFFFFFh**

**dwVal\_1** **DWORD** **0**

**dwVal\_2** **DWORD** **0**

**wVal**    **WORD**    **0**

**.code**

```
                  ; ESP Before| ESP After
PUSH 1           ; 0x0018FF84| 0x0018FF80
PUSH a           ; 0x0018FF80| 0x0018FF7E
PUSH b           ; 0x0018FF7E| 0x0018FF7A
```

```
; Now, we pop the values off the stack:
```

```
                  ;ESP Before | ESP After
POP dwVal_1 ;0x0018FF7A | 0x0018FF7E
POP wVal    ;0x0018FF7E | 0x0018FF80
POP dwVal_2 ;0x0018FF80 | 0x0018FF84
```

```
; In a nested loop, save and restore the
outer loop's count:
```

**.data**

**multiplicand** **DWORD** **2**

**.code**

**MOV** **eax, 1**       ; **EAX = 1**

**MOV** **ecx, 3**

**outer\_loop:**

**PUSH** **ecx**       ; **save**

**MOV** **ecx, 2**

**inner\_loop:**

**MUL** **multiplicand**

**LOOP** **inner\_loop**

**POP** **ecx**       ; **restore**

**LOOP** **outer\_loop**

```
                  ; EAX = 64
```

```
; Please also look at the nested loop
example in LOOP instruction.
```

		State Diagram		Loaded to	Contains
Instruction	Step	Stack pointer	Stack offset		
<b>pop</b> dwVal_1	store 1		0x0018FF80	1	
			0x0018FF7E	ABCDh	
	ESP -->		0x0018FF7A	FFFFFFFFh	dwVal_1 FFFFFFFFh
<b>increment</b> 2			0x0018FF80	1	
		ESP -->	0x0018FF7E	ABCDh	
<b>pop</b> wVal	store 1		0x0018FF80	1	
			0x0018FF7E	ABCDh	wVal ABCDh
	ESP -->				
<b>increment</b> 2			0x0018FF80	1	
		ESP -->			
<b>pop</b> dwVal_2	store 1	ESP -->	0x0018FF80	1	dwVal_2 1
	<b>increment</b> 2	ESP -->	0x0018FF84	?	

## POPAD - Pop All GP Registers

Pops the top 32 BYTES from the top of the stack into eight general-purpose registers in the following order: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX. The instruction pops off the stack the same registers' values that PUSHAD pushed on the stack, but in reverse order. For ESP, POPAD dismisses the value saved on the stack; instead, ESP is incremented when each register is loaded.

### Instruction format

POPAD

### EFLAGS Register Unchanged

```
MOV  eax, 1
MOV  ecx, 2
MOV  edx, 3
MOV  ebx, 4
; esp = 0x0018FF80
; ebp = 0x0018FF94
; esi = 0x00401005
; edi = 0x00401005

CALL saveAndRestore ; ESP=0x0018FF80
...
saveAndRestore PROC
    PUSHAD            ; ESP=0x0018FF60
                      ; (1) save

    MOV  ebp,esp
    MOV  eax,0Ah
    MOV  ecx,0Bh
    MOV  edx,0Ch
    MOV  ebx,0Dh
    MOV  esi,ebp
    ADD  esi,16 ; ESI=0x0018FF70
    MOV  edi,ebp
    ADD  edi,12 ; EDI=0x0018FF80

    POPAD            ; ESP=0x0018FF80
                      ; (2) restore

    RET
saveAndRestore ENDP
```

Register	Before the procedure call	PUSHAD executes	Procedure changes values	POPAD executes
ESP	0018FF80h	0018FF60h	0018FF60h	0018FF80h
EAX	00000001h	00000001h	0000000Ah	00000001h
ECX	00000002h	00000002h	0000000Bh	00000002h
EDX	00000003h	00000003h	0000000Ch	00000003h
EBX	00000004h	00000004h	0000000Dh	00000004h
ESP	0018FF80h	0018FF60h	0018FF60h	0018FF80h
EBP	0018FF94h	0018FF94h	0018FF60h	0018FF94h
ESI	00401005h	00401005h	0018FF70h	00401005h
EDI	00401005h	00401005h	0018FF80h	00401005h

## POPFD - Pop EFLAGS Register

Pops the top 4 BYTES of the stack into the EFLAGS (Status Flags) Register. Use to restore the flags after a procedure has been executed. This should be paired with the PUSHFD Instruction.

### Instruction format

**POPFD**

### EFLAGS changed

OF/OV	*	DF/UP	*	IF/EI	*	SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	*

\* changes

```
; usage example pushes the flags on the  
stack and immediately save them into a  
variable
```

```
.data
```

```
flagsBackup DWORD ?
```

```
.code
```

```
PUSHFD
```

```
; copy EFLAGS into a variable
```

```
POP flagsBackup
```

```
CALL systask
```

```
; restore the flags from the variable
```

```
PUSH flagsBackup
```

```
; copy the values into EFLAGS
```

```
POPFD
```

## PUSH - Push on Runtime Stack

Decrements the ESP value (the stack pointer) and then loads a source operand onto the stack's top. The ESP points to the stack's location. Pushing a DWORD register or memory (32-bit) operand decrements the offset stored in the ESP register by 4. Pushing a WORD register or memory (16-bit) operand decrements the ESP by 2. Pushing an immediate value (IA-32 architecture) decrements the ESP by 4. Using only the doublewords helps with keeping track of the ESP value and accessing values stored on stack.

### Instruction format

```
PUSH reg16/mem16
PUSH reg32/mem32
PUSH imm32
```

### EFLAGS Register Unchanged

```
; Push imm32, mem16, and mem32
```

```
.data
wVal WORD 0ABCDh
dwVal DWORD 0FFFFFFFFh
.code

; ESP Before | ESP After
; (Decrement) | (Store @)
PUSH 1 ; 0x0018FF84 | 0x0018FF80
PUSH wVal ; 0x0018FF80 | 0x0018FF7E
PUSH dwVal ; 0x0018FF7E | 0x0018FF7A
```

Instruction	Steps	Stack pointer	Value Stored @	Value Stored
push 1	1 decrement		0x0018FF84	??
	2 store	ESP -->	0x0018FF80	1
push wVal	1 decrement		0x0018FF80	1
	2 store	ESP -->	0x0018FF7E	ABCDh
push dwVal	1 decrement		0x0018FF80	1
	2 store	ESP -->	0x0018FF7A	FFFFFFFFh

```
; Push reg32 and reg16
```

```
; ESP Before | ESP After
; (Decrement) | (Store @)
MOV eax,12345678h
PUSH eax ; 0x0018FF7A | 0x0018FF76
MOV bx,0ABCDh
PUSH bx ; 0x0018FF76 | 0x0018FF74
```

## PUSHAD - Push All GP Registers

The instruction pushes all of the general-purpose registers onto the stack in the following order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. The ESP's value is the stack pointer's value before PUSHAD stores the EAX register. This single instruction is faster and requires fewer bytes than pushing each register one by one. PUSHAD (at the beginning) and POPAD (at the end) can save and restore registers in a procedure.

### Instruction format

PUSHAD

### EFLAGS Register Unchanged

```
; ...
MOV  eax, 1
MOV  ecx, 2
MOV  edx, 3
MOV  ebx, 4
; let esp = 0018FF80h
; let ebp = 0018FF94h
MOV  esi, 7
MOV  edi, 8

CALL saveAndRestore ; ESP=0x0018FF80

; ...

saveAndRestore PROC
    PUSHAD    ; ESP = 0x0018FF60.
               ; 20h = 32d = 8 * 4.
               ; saves 8 registers,
               ; 4 bytes each.

    MOV  eax, 0FFh ; EAX=0000.00FFh
    XOR  ebx, ebx  ; EBX=0
    ADD  ecx, edx   ; ECX=5
    SUB  edx, 0Ah   ; EDX=FFFF.FFF9h

    POPAD    ; ESP = 0x0018FF80.
               ; restores 8 registers.
RET
saveAndRestore ENDP
```

## PUSHFD - Push EFLAGS Register

Pushes the 4 BYTE EFLAGS (Status Flags) register onto the stack. Use to save the status flags before a procedure call; then restore them after the procedure has been executed. This should be paired with the POPFD Instruction.

### Instruction format

**PUSHFD**

### EFLAGS Register Unchanged

*; usage example -- saves the EFLAGS register before the task procedure is called.*

**PUSHFD**

**CALL** sysTask

**POPFD**

**sysTask PROC**

*; any sequence of instructions*

**RET**

**sysTask ENDP**



## OR – Bitwise (Logical) OR

Performs a bitwise (boolean) OR operation on each pair of the matching bits in the source operand and destination operand. Places the result in the destination operand. For each matching bit, if at least one of the corresponding bits in the operands has the value 1, the instruction sets the result to 1. If none of the corresponding bits in the operands have the value 1, the instruction sets the result to 0. The operands must be the same size.

### Syntax

**OR** destination, source

### Instruction format

**OR** reg, reg                      **OR** reg, mem  
**OR** reg, imm                    **OR** mem, reg  
**OR** mem, imm

### EFLAGS changed

OF/OV	0	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	?	PF/PE	*	CF/CY	0

\* changes, 1 sets, 0 clears, ? may change, (blank) unchanged

; OR instruction sets bits without affecting other bits. OR always clears the Carry and Overflow flags. The Sign, Zero, and Parity flags are set or clear according to the result placed in the destination operand.

; sets bits 3 and 4

```
MOV AL, 10000001b
OR AL, 10011001b ; AL = 1001.1001b
```

; If a value is OR'ed with itself/zero, the flags reveal information about this value.

```
MOV EAX, -1
OR EAX, EAX ; ZR=0, PL=1, value in EAX<0
```

```
MOV EAX, 0
OR EAX, EAX ; ZR=1, PL=0, value in EAX=0
```

```
MOV EAX, 1
OR EAX, 0 ; ZR=0, PL=0, value in EAX>0
```

## REP – Repeat String Operator

Repeats a string instruction a specified number of times and decrements the ECX register, which contains the count. The instruction repeats until the value in ECX is zero. REP can be combined with MOVS, LODS, and STOS instructions.

### Instruction format

```
REP MOVSB/MOVSW/MOVS
REP MOVS destination, source
```

```
REP LODSB/LODSW/LODS
REP LODS destination, source
```

```
REP STOSB/STOSW/STOS
REP STOS destination, source
```

### EFLAGS Register Unchanged

NOTE: The repeated instruction may change the status flags, but the prefix itself doesn't

#### ; Usage example 1

```
.data
str1 BYTE "0123456789"
str2 BYTE (LENGTHOF str1) DUP (0)

.code
CLD
MOV ESI,OFFSET str1 ; source "0123456789"
MOV EDI,OFFSET str2 ; dest* "0000000000"
MOV ECX,10           ; set counter to 10
REP MOVSB           ; execute MOVSB
                    ; if ECX is not 0

MOV EAX,ECX
CALL WriteDec       ; >>> 0

; str1: "0123456789"
; str2: "0123456789"

; *destination
```

#### ; Usage example 2 (Alternative)

```
.data
str1 BYTE "ABCDEFGHJIJ"
str2 BYTE (LENGTHOF str1) DUP ("Z")

.code
CLD
MOV esi,OFFSET str1 ; source "ABCDEFGHJIJ"
MOV edi,OFFSET str2 ; dest* "ZZZZZZZZZZ"
MOV ecx,10          ; count = 10
REP MOVS str2,str1

MOV eax,ecx
CALL WriteDec       ; >>> 0

; str1: "ABCDEFGHJIJ"
; str2: "ABCDEFGHJIJ"

; *destination
```

## REPE, REPZ – Repeat While (Equal, Zero)

Repeats a string instruction a specified number of times and decrements the ECX register, which contains the count. The instruction terminates when the value in ECX is zero or Zero Flag is clear. When both termination conditions coincide, use a JECXZ instruction to test the ECX or a JZ, JNZ, or JNE instructions to test the Zero Flag.

REPZ (repeat while zero) is a synonym for REPE (repeat while equal).

REPE/REPZ can be combined with CMPS and SCAS instructions, both of which modify the Zero Flag.

### Instruction format

```
REPE SCAS destination
REPE SCASB
REPE SCASW
```

```
REPZ SCAS destination
REPZ SCASB
REPZ SCASW
```

### EFLAGS Register Unchanged

NOTE: The repeated instruction may change the status flags, but the prefix itself doesn't

```
; Usage example 1
.data
array1 WORD 0Ah, 0Bh, 0Ch
array2 WORD 0Ah, 0Bh, 0Ch

.code
CLD
MOV esi, OFFSET array1
MOV edi, OFFSET array2
MOV ecx, LENGTHOF array1
REPE CMPSW ; repeat while ZF = 1

JE equal ; If ZF = 1, arrays are equal
JMP notEqual
equal:
MOV eax, 1
CALL WriteDec ; >>> 1
JMP continue
notEqual:
XOR eax, eax
CALL WriteDec

continue:
```

```
; Usage Example 2
.data
string1 BYTE "zero flag is clear",0
string2 BYTE "zero flag is clear",0

.code
CLD
MOV esi, OFFSET string1 first string
MOV edi, OFFSET string2 second string
MOV ecx, LENGTHOF string1
REPE CMPSB ; repeat while ZF = 1
JE equal
JMP notEqual
equal:
MOV eax, 1
CALL WriteDec
JMP continue
notEqual:
XOR eax, eax
CALL WriteDec ; >>> 0

continue:
```

## REPNE, REPNZ – Repeat While Not (Equal, Zero)

Repeats a string instruction a specified number of times and decrements the ECX register, which contains the count. The instruction terminates when the value in ECX is zero or Zero Flag is set. When both termination conditions coincide, use a JECXZ instruction to test the ECX or a JZ, JNZ, or JNE instructions to test the Zero Flag.

REPNZ (repeat while not zero) is a synonym for REPNE (repeat while not equal).

REPNE/REPNZ can be combined with CMPS and SCAS instructions, both of which modify the Zero Flag.

### Instruction format

```
REPNE SCAS destination
REPNE SCASB
REPNE SCASW
```

```
REPNZ SCAS destination
REPNZ SCASB
REPNZ SCASW
```

### EFLAGS Register Unchanged

NOTE: The repeated instruction may change the status flags, but the prefix itself doesn't

```
; Usage example 1
.data
target DWORD "y"
string BYTE \
"The five boxing wizards jump quickly",0

.code
MOV EAX,target ; AL = the look-up value
MOV EDI,OFFSET string
MOV ECX,LENGTHOF string
CLD
; repeat until ECX = 0 or target = [EDI]
REPNE SCASB

JE match ; if ZF = 1, jump to match
XOR EAX,EAX
CALL WriteDec ; Otherwise, output 0
JMP continue

match:
MOV EAX,1
CALL WriteDec

continue:

; >>> 1
```

```
; Usage Example 2
.data
target DWORD "."
string BYTE \
"The five boxing wizards jump quickly",0

.code
MOV EAX,target ; AL = the look-up value
MOV EDI,OFFSET string
MOV ECX,LENGTHOF string
CLD
; repeat until ECX = 0 or target = [EDI]
REPNZ SCASB

JE match ; if ZF = 1, jump to match
XOR EAX,EAX
CALL WriteDec ; Otherwise, output 0
JMP continue

match:
MOV EAX,1
CALL WriteDec

continue:

; >>> 0
```

## RET - Return from Procedure

Pops off the return address located on the top of the stack into the EIP register (the instruction pointer). Then returns to the instruction that immediately follows the CALL instruction. Accurately managing the stack is essential -- otherwise, RET may pop off incorrect value into the instruction pointer.

### Instruction format

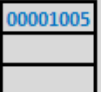
RET

### EFLAGS Register Unchanged

```
main PROC  
  
00001000 CALL subprocedure  
00001005 XOR  eax,  eax  
  
main ENDP  
  
subprocedure PROC  
  
00001100 XOR  ebx,  ebx  
  
        RET  
subprocedure ENDP
```

**CALL** 1) Pushes the offset of the instruction, which immediately follows it, onto the stack.

ESP --> 00001005

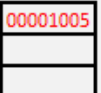


2) Loads the offset of subprocedure's first instruction statement into EIP.

EIP 00001100

**RET** 1) Pops the return address from the top of the stack into the EIP.

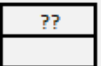
ESP --> 00001005



2) Returns to the instruction at offset that immediately follows the CALL.

EIP 00001005

ESP --> ??



## SCASB, SCASW, SCASD - Scan String

Compares the value loaded into the accumulator register to the contents of the memory location. EDI must contain the offset of this memory location. Sets status flags in EFLAGS register according to the search results. If the *direction flag* is clear, increments EDI; if the *direction flag* is set, decrements EDI. If values in the accumulator and memory location match, sets the Zero (ZF) flag.

Instruction	Look-up value loaded into	Memory Operand Size	Memory contents	EDI change
SCASB	AL	BYTE	[EDI]	EDI = EDI ± 1
SCASW	AX	WORD	[EDI]	EDI = EDI ± 2
SCASD	EAX	DWORD	[EDI]	EDI = EDI ± 4

Preceded by the REPNE prefix, the instruction scans the array or string until either the accumulator's value matches a value in memory or ECX equals zero. The zero flag is cleared if a match is not found.

Preceded by the REPE (or REPZ) prefix, the instruction scans the array or string while ECX is greater than zero, and the accumulator's value equals each subsequent memory value. The zero flag is set if all values match.

### Instruction format

SCASB  
SCASW  
SCASD

### EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	*

\* changes, (blank) unchanged

Usage Examples follow on next page

```

; Usage example 1
; Scans a string of characters until the
value is found.

.data
string BYTE \
"The five boxing wizards jump quickly",0
stringLen EQU LENGTHOF string

.code
CLD
MOV ecx, stringLen
LEA edi, string
MOV al, "i"
REPNE SCASB ; repeat while not equal
JCXZ notfound
MOV eax, [edi]
CALL WriteChar ; outputs the character
; coming after 'i'
; >>> v

JMP continue
notfound:
XOR eax, eax
CALL WriteDec

continue:

```

```

; Usage example 2a

.data
string BYTE "ab",0
stringLen EQU LENGTHOF string

.code
CLD
MOV ecx, stringLen
LEA edi, string ; load offset into EDI
MOV al, "a"
REPE SCASB ; repeat while equal

JZ allMatched
; if mismatch, output the memory's value

DEC edi ; EDI - 1
; points to the value
MOV eax, [edi] ; >>> 'b'
CALL WriteChar
JMP continue

allMatched:
MOV eax, 1
CALL WriteDec

continue:

```

```

; Usage example 2b

.data
string BYTE "aa"
stringLen EQU LENGTHOF string

.code
CLD
MOV ecx, stringLen
LEA edi, string
MOV al, "a"
REPE SCASB ; repeat while equal

JZ allMatched
; if mismatch, output the memory's value

DEC edi

MOV eax, [edi]
CALL WriteChar
JMP continue

allMatched:
MOV eax, 1 ; >>> 1
CALL WriteDec

continue:

```

## STD - Set Direction Flag

Sets the direction flag. String instructions decrement index registers (ESI and EDI) and process strings downward from high offset to low offset (right to left through the string in reverse direction).

### Instruction format

STD

### EFLAGS changed

OF/OV		DF/UP	1	IF/EI		SF/PL	
ZF/ZR		AF/AC		PF/PE		CF/CY	

*1 sets, (blank) unchanged*



## STOSB, STOSW, STOSD - Store String Data

Stores the register's contents into the memory location. EDI should contain the offset that points to the specific location in memory. If the direction flag is clear, EDI is incremented. If the direction flag is set, EDI is decremented. Combined with the REP prefix, STOS\* instructions can load the same value into all array elements.

Instruction	Loads from	Destination operand	EDI change	Equivalent Instructions (DF=0)	Equivalent Instructions (DF=1)
STOSB	AL	BYTE	EDI = EDI ± 1	mov [edi], al inc edi	mov [edi], al dec edi
STOSW	AX	WORD	EDI = EDI ± 2	mov [esi], ax add edi, 2	mov [esi], ax sub edi, 2
STOSD	EAX	DWORD	EDI = EDI ± 4	mov [edi], eax add edi, 4	mov [edi], eax sub edi, 4

### Instruction format

```
stosb
stosw
stosd
```

### EFLAGS Register Unchanged

```
; Loads the value 0fh into each element
; of an array of size 100 bytes
```

```
.data
array BYTE 100 DUP(?)
```

```
.code
MOV al, 0Fh
MOV edi, OFFSET array
MOV ecx, LENGTHOF array
CLD
REP STOSB ; stores 100 BYTES
```

```
; Reduces the number of iterations
; from 100 to 25.
```

```
ARRAY_SIZE = 100
.data
array BYTE ARRAY_SIZE DUP(0)
len EQU ARRAY_SIZE / 4
```

```
.code
MOV eax, 0FFFFFFFFh
MOV edi, OFFSET array
MOV ecx, len
CLD
REP STOSD ; stores 25 DWORDs
```

# SUB - Integer Subtraction

Subtracts the source operand from the destination operand; can be used to subtract both signed and unsigned integers. Only one operand can be a memory operand, and both operands must be the same size. The source operand is unchanged by the operation. The result of the operation is stored in the destination operand.

## Syntax

**SUB** destination, source

## Instruction formats

**SUB** reg, reg      **SUB** reg, imm  
**SUB** reg, mem      **SUB** mem, reg  
**SUB** mem, imm

## EFLAGS changed

OF/OV	*	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	*	PF/PE	*	CF/CY	*

\* changes, (blank) unchanged

## ; Usage examples

```
.data
dwordVal  DWORD 0ABCD1234h
.code
MOV  EAX, 0ABCD1234h
MOV  EBX, 0ABCD0000h
SUB  EAX, EBX           ;EAX = 0000.1234h
SUB  EAX, 34h           ;EAX = 0000.1200h
SUB  dwordVal, EAX      ;dwordVal = ABCD.0034h
SUB  dwordVal, 34h      ;dwordVal = ABCD.0000h
SUB  EBX, dwordVal      ; EBX = 0000.0000h
```

## ; Use PTR operator to perform calculations on data in different sizes

```
.data
byteVal  BYTE 0FFh
.code
MOV  EAX, 0FFFFFFFFh
SUB  EAX, DWORD PTR byteVal ;EAX= FFFF.FF00h
MOV  BX, 0FFFFh
SUB  BX, WORD PTR byteVal ; BX = FF00h
```

## ; Unsigned operations. Zero flag.

; 1-1 = 0, so the 0 in the  
; destination operand sets the Zero Flag.

```
MOV AL, 1
SUB AL, 1 ; AL = 0, ZR = 1
```

; 2-1=1, 1 in the destination operand  
clears the Zero Flag.

```
MOV AL, 2
SUB AL, 1 ; AL = 1, ZR = 0
```

## ; Signed operations. Zero Flag.

; (-1)-(-1) = 0, so the 0 in the  
; destination operand sets the Zero Flag.

```
MOV EBX, -1
SUB EBX, -1 ; EBX = 0, ZR = 1
```

; -2-1=-3, -3 in the destination operand  
clears the Zero Flag.

```
MOV EBX, -2
SUB EBX, 1 ; EBX = -3, ZR = 0
```

## Signed operations. Overflow flag.

; The overflow flag is set when the result  
underflows the destination operand

```
XOR  EAX, EAX
MOV  AL, -128 ;AL = 80h = -128
SUB  AL, 1    ;AL = 7Fh = +127, OV=1
```

```
XOR  EAX, EAX
MOV  AX, -32768 ;AL = 8000h = -32768
SUB  AX, 1      ;AL = 7FFFh = +32767, OV=1
```

```
XOR  EAX, EAX
MOV  EAX, -2147483648
SUB  EAX, 1 ;OV = 1
```

## Unsigned operations. Parity Flag.

; An even number of 1 bits in the least  
significant byte of the destination sets the  
flag. Works on 8 bits only.

```
MOV AL, 10000011b ; AL=1000011b, PE = 0
SUB AL, 00000010b ; AL=1000001b, PE = 1
```

; Unsigned operations. Carry flag!

; A larger unsigned integer is subtracted from a smaller one.

```
MOV AL, 127 ; AL=7Fh
SUB AL, 128 ; AL=FFh=-1, CY=1
```

	0 1 1 1 1 1 1 1	127
+	1 0 0 0 0 0 0 0	(-128)
CF	1	1 1 1 1 1 1 1 1

; Signed operations. Overflow Flag!

; A signed source is subtracted from a signed destination.  
; The output is invalid!

```
MOV AL, -128 ; AL=80h
SUB AL, 1 ; AL=7Fh=+127, OV=1
```

	1 0 0 0 0 0 0 0	(-128)
-	0 0 0 0 0 0 0 1	1
	0 1 1 1 1 1 1 1	

; Unsigned operations. Auxiliary Carry Flag

; A borrow from the high nibble to the low nibble occurs.

```
XOR EAX, EAX ; Zero out EAX
MOV AL, 16 ; AL = 0001 0000b
SUB AL, 8 ; AL = 0000 1000b
```

bit position	7 6 5 4 3 2 1 0		
	high nibble	low nibble	
	0 0 0 1	0 0 0 0	16
-	0 0 0 0	1 0 0 0	8
	0 0 0 0	1 0 0 0	8

recall that

	2 d	1 0 b	borrow 1	10b = 1b + 1b
-	1 d	0 1 b		10b - 1b = 1b
	1 d	1 b		

\* d = decimal, b = binary

; Signed Operations. Sign Flag.

; When the result of a signed arithmetic operation is negative, the Sign Flag is set; it is clear otherwise.

```
MOV AL, 127 ; AL = 7Fh = 127
SUB AL, 2 ; AL = 7Dh = 125, PL = 0
SUB AL, 126 ; AL = FFh = -1, PL = 1
```

	0 1 1 1 1 1 1 1	127		
-	0 0 0 0 0 0 1 0	2		
SF	0	0	1 1 1 1 1 0 1	125
	0 1 1 1 1 1 0 1	125		
-	0 1 1 1 1 1 1 0	126		
SF	1	1	1 1 1 1 1 1 1	-1

The sign flag is the copy of the destination operand's most significant bit.

## XCHG – Exchange Values

Exchanges data values in the source and destination operands. The contents can be exchanged between two general-purpose registers, or between a register and a location in memory, but not directly between a memory location and another memory location.

The operands must match in size, cannot both be memory operands. The instruction pointer register (EIP) cannot be a destination.

### Syntax

**XCHG** destination, source

### Instruction format

**XCHG** reg, reg                      **XCHG** reg, mem  
**XCHG** mem, reg

### EFLAGS Register Unchanged

; reg8, reg8

```
MOV AL, 0      ; AL=00h
MOV BL, 0ABh   ; BL=ABh
XCHG AL, BL    ; AL=ABh, BL=00h
```

; reg8, mem8            ; mem8, reg8

```
.data
byteVal BYTE 255

.code
MOV AL, 0      ; AL=0, byteVal=255
XCHG byteVal, AL ; AL=255, byteVal=0
XCHG AL, byteVal ; AL=0, byteVal=255
```

; reg16, reg16

```
MOV AX, 0      ; AX=0000h
MOV BX, 0ABCDh ; AX=0000h, BX=ABCDh
XCHG AX, BX    ; AX=ABCDh, BX=0000h
```

; reg16, mem16            ; mem32, reg32

```
.data
wordVal WORD 65535 ;FFFFh
dwordVal DWORD 4294967295 ;FFFF.FFFFh

.code
MOV EAX, 0ABCD0000h ; AX=0000h
XCHG AX, wordVal    ; AX=FFFFh, wordVal=0
XCHG dwordVal, EAX
; EAX=FFFF.FFFFh, dwordVal=ABCD.FFFFh
```

; reg32, reg32

```
MOV EAX, 0      ; EAX = 0h
MOV EBX, 0ABCD1234h
XCHG BL, BH     ; EBX=ABCD.3412h
XCHG EAX, EBX   ; AX=ABCD.3412h, EBX=0h
```

; use mov and xchg to swap data between two locations in memory

```
.data
value0 DWORD 1
value1 DWORD 0

.code
MOV EAX, value0 ; EAX=1, value0=1
XCHG EAX, value1 ; EAX=0, value1=1
MOV value0, EAX ; EAX=0, value0=0
```

## XOR – Bitwise (Logical) *Exclusive OR*

Performs a bitwise (boolean) XOR operation on each pair of the matching bits in the source operand and destination operand. Places the result in the destination operand. For each matching bit, if both corresponding bits are the same (both are either 1 or 0), the result is 0. If either, but not both, of the corresponding bits has the value 1, the result is 1. The operands must be the same size.

This instruction may also be used as the most efficient method to set a register contents to 0, by XOR'ing a register with.

### Syntax

**XOR** destination, source

### Instruction format

**XOR** reg, reg

**XOR** reg, mem

**XOR** reg, imm

**XOR** mem, reg

**XOR** mem, imm

### EFLAGS changed

OF/OV	0	DF/UP		IF/EI		SF/PL	*
ZF/ZR	*	AF/AC	?	PF/PE	*	CF/CY	0

\* changes, 1 sets, 0 clears, ? may change, (blank) unchanged

; Toggle the value of specific bits, i.e., reverse them from their current settings.

; use 1 for bit position to be toggled

; use 0 for bit position to remain unchanged

**MOV** AL, 00110111b

**XOR** AL, 10101101b

; AL = 10011010b

; The most efficient way to clear out a register, which is faster than it would take to transfer 0 using the MOV instruction.

**MOV** EAX, 4294967295 ;EAX = FFFF.FFFFh

**XOR** EAX, EAX ;EAX = 0000.0000h

# Floating Point Instructions

## **FINIT** – Initialize FPU

Call before using any other FPU instructions in a program.

- Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states.
- Sets the FPU control word to 037Fh
  - Round to nearest
  - All exceptions masked
  - 64-bit precision
- Clears the status word
  - Clears all exception flags
  - Set TOP to 0
- Does not change the FPU stack registers but tags them with 11b (empty).
- Clears the FPU instruction and data pointers.

### *Exception Mask Bits*

Bits 0-5 of the FPU Control Word mask the 6 floating-point exception flags in the x86 FPU status word. Setting one of these mask bits prevents the corresponding floating-point exception from being generated.

FPUControlWord	037FH
FPUStatusWord	0
FPUTagWord	0FFFFH
FPUDataPointer	0
FPUInstructionPointer	0
FPULastInstructionOpcode	0
FPU Flags C0, C1, C2, C3	all set to 0

### **Syntax**

**FINIT**

## FLD – Load Floating Point value

Push a floating-point value (memory operand or selected FPU register) – onto the FPU stack.

### Syntax

**FLD** source

### Instruction formats

```
FLD mem32fp      ; a single-precision 32-bit short real
FLD mem64fp      ; a double-precision 64-bit long real
FLD mem80fp      ; a double-extended-precision 80-bit extended real
FLD ST(i)        ; a selected FPU register (0-7)
```

### Usage examples

.data

```
shortReal      REAL4    271.401
longReal       REAL8    808.209
extendedReal   REAL10   111.123456789
```

```
singlePrecision REAL4    ?
doublePrecision REAL8    ?
```

.code

**FINIT**

```
FLD shortReal
FLD longReal
FLD extendedReal      ; The FPU Stack. State Diagram:
                      ; ST(0) = +111.12345678900000
                      ; ST(1) = +808.20899999999994
                      ; ST(2) = +271.40100097656250

FST singlePrecision   ; 111.123459
                      ; (!) FST rounds the significand to the width of the destination

FST doublePrecision   ; 111.12345678900000

FST ST(1)             ; The FPU Stack. State Diagram:
                      ; ST(0) = +111.12345678900000
                      ; ST(1) = +111.12345678900000
                      ; ST(2) = +271.40100097656250
```

## **FILD** - Convert Integer to Float and Load Value

Converts the signed-integer source operand, stored in memory (a word, doubleword, or quadword integer) into a double extended-precision floating-point format. Then pushes the value onto the FPU register stack.

### **Syntax**

**FILD** source

### **Instruction formats**

<b>FILD</b>	m16int	; a signed word integer operand in memory
<b>FILD</b>	m32int	; a signed doubleword integer operand in memory
<b>FILD</b>	m64int	; a quadword integer operand in memory

### **Usage examples**

```
.data
sWordVal      SWORD    -32768                ; -2^15
sDwordVal     SDWORD   -2147483648           ; -2^31
qWordVal      QWORD    -9223372036854775808   ; -2^63
qWordVal1     QWORD    4294967295             ; +2^32-1

.code

FINIT

FILD          sWordVal      ; ST(0) = -32768.000000000000
FILD          sDwordVal     ; ST(0) = -2.1474836480000000 e+0009
FILD          qWordVal      ; ST(0) = -9.2233720368547758 e+0018
FILD          qWordVal1     ; ST(0) = +4.2949672950000000 e+0009
                                   ; ST(0): the top element of the FPU register stack.
```



## FST – Store Floating Point Value

Copies the value from ST(0) to the destination operand. The destination operand can be a location in memory or a register in the FPU register stack. Converts the value, which goes to the destination operand, to single-precision or double-precision floating-point format. Does not pop the stack. ST(i) refers to any ST register

### Syntax

**FST** destination

### Instruction formats

**FST** m32fp ; a single-precision floating point memory operand  
**FST** m64fp ; a double-precision floating point memory operand  
**FST** ST(i) ; the i-th element from the top of the FPU register stack (0 through 7)

### Usage examples

.data

shortReal        **REAL4**    271.401  
longReal        **REAL8**    808.209  
extendedReal    **REAL10** 111.123456789

singlePrecision **REAL4**    ?  
doublePrecision **REAL8**    ?

.code

**FINIT**

**FLD** shortReal  
**FLD** longReal  
**FLD** extendedReal ; The FPU Stack. State Diagram:  
; ST(0) = +111.123456789...  
; ST(1) = +808.20899...  
; ST(2) = +271.4010...

**FST** singlePrecision ; 111.123459  
; (!) FST rounds the significand to the width of the destination

**FST** doublePrecision ; 111.12345678900000

**FST** ST(1) ; The FPU Stack. State Diagram:  
; ST(0) = +111.12345678900000  
; ST(1) = +111.12345678900000  
; ST(2) = +271.40100097656250

## FSTP – Store Floating Point Value and Pop

Copies the value in ST(0) to the memory destination. Then pops the register value off the stack. Can also store values in memory in double extended-precision floating-point format. The processor increments the stack pointer by 1. ST(i) refers to any ST register

### Syntax

**FSTP** destination

### Instruction formats

**FSTP** m32fp ; a single-precision floating point memory operand  
**FSTP** m64fp ; a double-precision floating point memory operand  
**FSTP** m80fp ; a double extended-precision floating point memory operand  
**FSTP** ST(i) ; the i-th element from the top of the FPU register stack (0 through 7)

### Usage examples

.data

shortReal           **REAL4**    271.401  
longReal            **REAL8**    808.209  
extendedReal        **REAL10** 111.1234567890

singlePrecision     **REAL4**    ?  
doublePrecision     **REAL8**    ?  
dExtendedPrecision **REAL10** ?

.code

**FINIT**

**FLD** shortReal  
**FLD** longReal  
**FLD** extendedReal ; The FPU Stack. State Diagram:  
; ST(0) = +111.12345678900000  
; ST(1) = +808.20899999999994  
; ST(2) = +271.40100097656250

**FSTP** singlePrecision ; 111.123459  
; The FPU Stack. State Diagram:  
; ST(0) = +808.20899999999994  
; ST(1) = +271.40100097656250  
; ST(2) = +0.0000000000000000

**FSTP** doublePrecision ; 808.20899999999995  
; (!) FSTP rounds the significand to the width of the destination  
; The FPU Stack. State Diagram:  
; ST(0) = +271.40100097656250  
; ST(1) = +0.0  
; ST(2) = +0.0

## FADD – FPU Addition

Adds the source operand to the destination operand. Stores the sum in the destination operand. The destination operand is an FPU register. The source operand is either an FPU register or a location in memory. ST(i) refers to any ST register  
Also available as a 0-address operation.

### Syntax

```
FADD          ; 0-address operation
              ; 1. adds ST(1) to ST(0),
              ; 2. stores result in ST(1),
              ; 3. pops ST(0) off the stack.
FADD source          ; adds the contents of a source to ST(0)
FADD destination, source ; adds the contents of source to destination
```

### Instruction formats

```
FADD          ; 0-address operation

FADD m32fp          ; a single-precision float operand or word integer
                   ; ST(0) = ST(0) + m32fp

FADD m64fp          ; a double-precision float operand or doubleword integer
                   ; ST(0) = ST(0) + m64fp

FADD ST(0), ST(i)   ; ST(i) i-th element from the top of the FPU register stack (0-7)
                   ; ST(0) = ST(0) + ST(i)

FADD ST(i), ST(0)   ; ST(i) = ST(i) + ST(0)
```

### Usage examples

```
.data
shortReal      REAL4  271.401
longReal       REAL8  808.209
longReal2      REAL8  909.111

.code

FINIT

FLD  shortReal
FLD  longReal      ; ST(0) = +808.2089...
                   ; ST(1) = +271.401...

FADD          ; ST(0) = +1079.610...

FADD  longReal2    ; ST(0) = +1988.721...

FADD  ST(0), ST(0) ; ST(0) = +3977.442...
```

## FSUB – FPU Subtraction

Subtracts the source operand from the destination operand. Then stores the result in the destination operand. The destination operand is an FPU register. The source operand is either an FPU register or a location in memory. ST(i) refers to any ST register  
Also available as a 0-address operation.

### Syntax

```
FSUB          ; 0-address operation
              ; 1. subtracts ST(0) from ST(1),
              ; 2. stores result in ST(1),
              ; 3. pops ST(0) off the stack.
FADD source          ; subtracts the contents of a source from ST(0)
FADD destination, source ; subtracts the contents of source from destination
```

### Instruction formats

```
FSUB          ; 0-address operation

FSUB m32fp          ; a single-precision floating-point operand or word integer
                  ; ST(0) = ST(0) - m32fp

FSUB m64fp          ; a double-precision floating-point operand or doubleword integer
                  ; ST(0) = ST(0) - m64fp

FSUB ST(0), ST(i)   ; ST(0) = ST(0) - ST(i)
FSUB ST(i), ST(0)   ; ST(i) = ST(i) - ST(0)
```

### Usage examples

```
.data
longReal      REAL8  999.999
longReal2     REAL8  111.111

.code

FINIT

FLD longReal
FLD longReal2      ; ST(0) = +111.111...
                  ; ST(1) = +999.999...
FSUB          ; ST(0) = +888.888...
FSUB longReal2 ; ST(0) = +777.777...
```

## FMUL – FPU Multiplication

Multiplies the destination and source operands. Stores the product in the destination operand. The destination operand is an FPU register. The source operand is either an FPU register or a location in memory. ST(i) refers to any ST register  
Also available as a 0-address operation.

### Syntax

```
FMUL          ; 0-address operation
              ; 1. Multiplies ST(1) by ST(0),
              ; 2. stores result in ST(1),
              ; 3. pops ST(0) off the stack.
FMUL source    ; Multiplies ST(0) by the contents of source
FMUL destination, source ; Multiplies contents of destination by contents of source
```

### Instruction formats

```
FMUL          ; 0-address operation (see above)
FMUL m32fp     ; ST(0) = ST(0) x a single-precision float, or a word integer
FMUL m64fp     ; ST(0) = ST(0) x a double-precision float, or a doubleword integer
FMUL ST(0), ST(i) ; ST(0) = ST(0) x ST(i)
FMUL ST(i), ST(0) ; ST(0) = ST(0) x ST(i)
```

### Usage examples

```
.data
shortReal REAL4 1.11
longReal REAL8 800.9
longReal2 REAL8 244.504865

.code

FINIT

FLD shortReal
FLD longReal      ; ST(0) = +800.89
                  ; ST(1) = +1.11
FMUL              ; ST(0) = +888.999
                  ; ST(1) = +0.00000
FMUL shortReal    ; ST(0) = +986.78889

FLD longReal2
FLD shortReal     ; ST(0) = +1.11
                  ; ST(1) = +244.504865
                  ; ST(2) = +986.78889
FMUL ST(1), ST(0) ; ST(1) = +271.400
```

## FDIV – FPU Division

Divides the destination operand by the source operand. Stores the dividend in the destination operand. The destination operand is an FPU register. The source operand is either a register or a location in memory. ST(i) refers to any ST register  
Also available as a 0-address operation.

### Syntax

```
FDIV          ; 0-address operation
              ; 1. Divides ST(1) by ST(0),
              ; 2. stores result in ST(1),
              ; 3. pops ST(0) off the stack.
FDIV source   ; Divides ST(0) by the contents of source
FDIV destination, source ; Divides contents of destination by contents of source
```

### Instruction formats

```
FDIV
FDIV m32fp           ; ST(0) = ST(0) / a single-precision float, or a word integer
FDIV m64fp           ; ST(0) = ST(0) / a double-precision float, or a doubleword integer
FDIV ST(0), ST(i)    ; ST(0) = ST(0) / ST(i)
FDIV ST(i), ST(0)    ; ST(i) = ST(i) / ST(0)
```

### Usage examples

.data

```
shortReal  REAL4 0.10
shortReal2 REAL4 10.0
longReal   REAL8 111.222
```

.code

FINIT

```
FLD  longReal      ; ST(0) = +111.222
FDIV shortReal     ; ST(0) = +1112.22
FLD  shortReal2    ; ST(0) = +10.0000
FDIV ST(1), ST(0)  ; ST(1) = +111.222
```

## FCHS – Change Sign

Changes the sign of the Floating Point Value stored in the ST(0) register.

### Instruction formats

**FCHS**

### Usage examples

.data

```
shortReal REAL4 -80.86  
longReal REAL8 +9.10900
```

.code

**FINIT**

```
FLD    shortReal    ; ST(0) = -8.086  
FCHS                   ; ST(0) = +8.086
```

```
FLD    longReal     ; ST(0) = +9.10899  
FCHS                   ; ST(0) = -9.10899
```

## FABS – Absolute Value

Clears the sign bit of the Floating Point Value stored in ST(0) and creates its absolute value.

### Instruction formats

**FABS**

### Usage examples

.data

```
shortReal REAL4 -80.86  
longReal REAL8 +9.10900
```

.code

**FINIT**

```
FLD    shortReal    ; ST(0) = -80.86  
FABS                   ; ST(0) = +80.86
```

```
FLD    longReal     ; ST(0) = +9.10899  
FABS                   ; ST(0) = +9.10899
```