80x86 Instruction Reference

Instructions are listed in this reference in this general order. Instructions that are available in the 32-bit machines only are listed in lowercase.

- DATA MANIPULATION INSTRUCTIONS
  - Moving Data: MOV, PUSH, PUSHA, POP, POPA, pushad, popad, XCHG
  - I/O: IN, OUT, INS, OUTS
  - Address Loading: LEA, LDS/LES/lfs/lgs/lss
  - Flag Transfer: LAHF, SAHF, PUSHF/,pushfd, POPF/popfd

- ARITHMETIC AND COMPARISON
  - Addition: ADC, ADD, INC
  - Subtraction and Comparison: CMP, DEC, NEG, SBB, SUB
  - Multiplication: MUL, IMUL, IMUL 2-3 operands
  - Division: DIV, IDIV
  - Sign Extension: CBW, CWD, cwde, cdq, movsx, movzx
  - BCD Arithmetic: DAA, DAS, AAA, AAS, AAM, AAD

- BIT AND LOGICAL OPERATIONS
  - Logical: AND, TEST, OR, XOR, NOT
  - Shifts: SAL, SHL, SAR, SHR, shld, shrd
  - Rotations: RCL, RCR, ROL, ROR
  - Scan: bsf, bsr
  - Test: bt, btr, bts, btc

- CHARACTER OPERATIONS
  - Table Lookup: XLAT
  - Block (String) Moves: MOVS, MOVSB, MOVSW, LODS, STOS
  - Block (String) Comparison: CMPS, CMPSB, CMPSW
  - Search: SCAS, SCASB, SCASW
  - Repetition Prefixes: REP, REPE, REPZ, REPNE, REPNZ

- CONTROL FLOW
  - Jumps: JMP
  - Subroutines: CALL, RET
  - Conditionals: JA, JAE, JB, JBE, JC, JCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, jecxz
  - Loops: LOOP, LOOPE, LOOPNE, loopd, looped, loopned
  - Interrupts: INT, INTO, IRET
  - Flag Control: CLC, CLD, CLI, CMC, STC, STD, STI
  - Conditional Set: setcc

- MISCELLANEOUS
  - Space Filling: NOP
  - Miscellaneous: bswap, BOUND, ENTER, LEAVE
  - Process Synchronization: xadd, cmpxchg, cmpxchg8b
  - External Synchronization: ESC, HLT, LOCK, WAIT
  - Systems Programming (Not discussed): arpl, clts, cpuid, invd, invlpg, lar, lgdt, lidt, lldt, lmsw, lsl, ltr, rdsmr, rsm. sgdt, sidt, sldt, smsw, verr, verw, wbinvd, wrmsr

Notes:
Processor models listed in square brackets (e.g., [386]) indicate the earliest processor on which the instruction or a particular operand is available. Some syntax variations and operands are shown with a processor in parentheses; this also indicates the earliest processor for the syntax or operand.

MOV (MOVe)

Purpose:
Used to copy values from one place to another. Many other architectures use LOAD and STORE instruction separately.

Syntax:
MOV dest, source

Semantics:
dest <- source; source unaltered

Flags:
ODITSZAPC unchanged

Operands:

reg,reg	mem,reg	reg,mem
reg,imm	mem,imm	special reg,reg (80386)
reg16,segreg	mem16,segreg	reg,special reg (80386)
segreg,reg16	segreg,mem16	

Notes:
1. Cannot MOVe an immediate value into a segment register

PUSH/PUSHD [386]

Purpose:
Used to store data on the stack

Syntax:
PUSH operand

Semantics:
1. SP <- SP-2 (esp <- esp-2 or esp-4)
2. [SS:SP] <- operand ([ss:esp] <- operand)

Flags:
ODITSZAPC unchanged

Operands:

reg16	mem16	segreg	imm (80186)
reg32 (386)		mem32 (386)	

Notes:
1. Can only push words or dwords (80386)
2. PUSHD syntax only necessary if instruction is otherwise ambiguous, e.g., push eax is OK but pushd 42 would be necessary for push imm.

POP/ POPD [386]

Purpose:
Retrieve data from the stack

Syntax:
POP dest

Semantics:
1. dest <- [SS:SP] (dest <- [ss:esp])
2. SP <- SP + 2 (esp <- esp+2 or esp+4)

Flags:
ODITSZAPC unchanged

Operands:

reg16	mem16	segreg
reg32 (386)	mem32 (386)	

Notes on PUSH and POP:
1. SP (esp) is decremented as stack grows
2. SP (esp) always points to last item on stack, not to the next available word (except when stack is empty)
3. PUSH and POP always work with words or dwords. To PUSH a byte it has to be extended to a word
4. CALL, INT, RET and IRET also affect the stack

PUSHA/POPA [386] PUSHAD/POPAD [386]

Purpose:
Save/retrieve all GP registers

Syntax:
PUSHA POPA PUSHAD POPAD

Semantics:
1. SP <- SP-16 (pusha) esp <- esp-32 (pushad)
2. AX,BX,CX,DX,BP,SP,SI,DI pushed in order (32 bit regs same order)
3. POPA/POPAD restores registers and adds to SP/esp

Flags:
ODITSZAPC unchanged

Notes:
1. PUSHA is generally more efficient only when at least 5 registers have to be pushed on the stack. Otherwise individual PUSHes are more important.
2. Value of SP on stack can be used to distinguish 8086 from later processors

XCHG (eXCHanGe)

Purpose:
Exchange the contents of two registers, or one register and memory
Syntax:
XCHG dest1,dest2
Semantics:
dest1 <- dest2
dest2 <- dest1
Flags:
ODITSZAPC unchanged
operands
reg,reg mem,reg reg,mem
Notes:
1. At least one operand must be in a general register.
2. To perform an XCHG using MOV requires three instructions.

```
MOV TEMP,AX
MOV AX,BX
MOV BX,TEMP
```

IN and OUT

IN (INput from port)

Purpose:
Input a byte or word from an I/O port.
Syntax:

```
IN AL,port
OR:  IN AX,port
```

Semantics:
AL <- port OR AX <- port
Flags:
ODITSZAPC unchanged
Operands:
port can be immediate or DX
Notes:
Ports are numbered 0000H through FFFFH, although most PCs only use address 0-3FFH. Port can be specified either as an immediate operand or can be specified in DX.

OUT (OUTput to port)

Purpose:
Output a byte or word from an I/O port.
Syntax:

```
OUT port,AL
OR:  OUT port,AX
```

Semantics:
port <- AL OR port <- AX
Flags:
ODITSZAPC unchanged
Operands:
port can be immediate or can be in DX
Notes:
See IN above.

INS [186]/INSB/INSW/INSD[386] Input from port to string
OUTS [186]/OUTSB/OUTSW/OUTSD[386] Output from string to port

Purpose:
Input or output a series of bytes, words, or dwords (386) from/to an I/O port. This instruction operates in the same way as other String Instructions. Operands are implied.
Syntax:
INSB INSW INSD OUTSB OUTSW OUTSD
Semantics:
INSB, INSW, INSD: es:[di] or es:[edi] <- port
di or edi incremented or decremented per DF and operand size. **See String Instructions**
OUTSB, OUTSW, OUTSD: port <- ds:[si] or ds:[esi]
si or esi incremented or decremented per DF and operand size. **See String Instructions**
Flags:
ODITSZAPC unchanged
Operands:
port is specified in DX
dest (INS) is es:[di] or es:[edi]; source (OUTSB) is ds:[si] or ds:[esi]

Address Transfer Instructions
LEA (Load Effective Address)

Purpose:
    Loads address of operand into a register.
Syntax:
    LEA dest, operand
Semantics:
    dest <- [operand]
Flags:
    ODITSZAPC   unchanged
Operands:
    index reg, mem
Notes:
1. Rarely used in simplest form. MOV dest, OFFSET mem is one byte shorter and generally preferred. Note that MOV AX,BX is equivalent to LEA AX,[BX]
2. When used with an index register (e.g., LEA SI,[SI+4]) LEA can be used to address arithmetic without affecting the flags. Note that the square brackets in this case do not indicate a memory reference. The instruction above adds 4 to SI.

LDS (Load Far Pointer into DS)
LES (Load Far Pointer into ES)
LSS/LFS/LGS (Load Far Pointer into SS/FS/GS) [386]

Purpose:
    Load DS or ES and a register or memory location simultaneously with segment and offset.
Syntax:
    LDS dest, source   LSS dest, source   LGS dest, source
    LES dest, source   LFS dest, source
Semantics:
    DS <- SEGMENT source & dest <- OFFSET source   (LDS)
    ES <- SEGMENT source & dest <- OFFSET source   (LES)
    Similarly for SS, FS, and SS on 386+ processors
Flags:
    ODITSZAPC   unchanged
Operands:
    reg, far_mem

Flag Transfer: LAHF, SAHF, PUSHF/PUSHFD, POPF/POPFD
LAHF (Load AH from Flags)

Purpose:
    Loads AH from low byte of flags register (SZAPC flags)
Syntax:
    LAHF
Flags:
    ODITSZAPC   unchanged

SAHF (Store AH to Flags)

Purpose:
    Stores AH into low byte of flags register (SZAPC flags)
Syntax:
    SAHF
Flags:
    ODIT.....   unchanged
    ....SZAPC   copied from AH
Notes:
1. This instruction is the only way that the x86 can directly modify the SZAP bits in the flags register. Note that the ODIT bits cannot be accessed using these instructions.
2. S = bit 7; Z = bit 6; A = bit 4; P = bit 2; C = bit 0

PUSHF (PUSH Flags)/PUSHFD [386]

Purpose:
    Push a copy of the flags register onto the stack.
Syntax:
    PUSHF   PUSHFD
Semantics:
    (1) SP <- SP-2 (esp <- esp-2)
    (2) [SS:SP] <- Flags register (ss:esp <- flags)
    For pushfd EFLAGS is pushed and stack pointer adjusted by 4

POPF (POP Flags)/POPFD [386]

Purpose:
    Word at top of stack copied to Flags register.
Syntax:
    POPF
Semantics:
    (1) Flags <- [SS:SP]
    (2) SP <- SP + 2
Notes:
1. PUSHF and POPF can be used to save the state of the flags register before calling a subroutine.
2. POPF is the only instruction that allows modification of the Trap Flag, except that it is cleared when an INT is executed.
3. O = bit Bh; D = bit Ah; I = bit 9; T = bit 8. See SAHF above for other bits.

Addition: ADD, ADC

ADD

Purpose:
    Perform signed or unsigned addition
Syntax:
    ADD dest,source
Semantics:
    dest <- dest + source
Flags:

O...SZAPC	modified for sum
.DIT.....	unchanged

Operands:

reg,reg	mem,reg	reg,mem
reg,immed	mem,immed	

Notes:
1. Modifies ALL status flags

ADC (ADd with Carry)

Purpose:
    Adds carry flag to sum. Used to implement multiple precision arithmetic
Syntax:
    ADC dest,source
Semantics:
    dest <- dest + source + CF
Flags:

O...SZAPC	modified for sum
.DIT.....	unchanged

Operands:

reg,reg	mem,reg	reg,mem
reg,immed	mem,immed	

Example: 64-bit addition (X <- X+Y)

```
    LEA ESI,X
    LEA EDI,Y
    MOV EAX,[EDI]        ; EAX <- low order word
    MOV EDX,[EDI+2]      ; EDX <- high order word
    ADD [ESI],AX         ; add low order words
    ADC [ESI+2],DX       ; add high order words with carry from low
```

Subtraction: SUB and SBB

SUB (SUBtract)

Purpose:
    Perform signed or unsigned subtraction
Syntax:
    SUB dest,source
Semantics:
    dest <- dest - source
Flags:

O...SZAPC	modified for difference
.DIT.....	unchanged

Operands:

reg,reg	mem,reg	reg,mem
reg,immed	mem,immed	

Notes:
1. Remember that addition is commutative but subtraction is not
2. Modifies ALL status flags

SBB (SUBtract with Borrow)

Purpose:
    Multiple precision arithmetic
Syntax:
    SBB dest,source
Semantics:
    dest <- dest - source - CF
Flags:
    O...SZAPC   modified for difference
    .DIT.....   unchanged
Operands:
    reg,reg   mem,reg   reg,mem   reg,immed   mem,immed

CMP (CoMPare)

Purpose:
    Compare (CMP) is a subtraction without storage of the result that affects flags only. Compares two operands
Syntax:
    CMP leftOp, rightOp
Semantics:
    compute leftOp - rightOp
Flags:
    O...SZAPC   modified for difference
    .DIT.....   unchanged
Operands:
    reg,reg   mem,reg   reg,mem   reg,immed   mem,immed
Notes:
    This instruction is normally followed by conditional jump.

INC (INCrement)

Purpose:
    Increment register or memory by 1
Syntax:
    INC dest
Semantics:
    dest <- dest + 1
Flags:
    O...SZAP.   modified for sum
    .DIT....C   unchanged
Operands:
    reg   mem

DEC (DECrement)

Purpose:
    Decrement register or memory by 1
Syntax:
    DEC dest
Semantics:
    dest <- dest - 1
Flags:
    O...SZAP.   modified for difference
    .DIT....C   unchanged
Operands:
    reg   mem
Notes:
    INC and DEC do NOT affect the Carry flag and therefore are not completely equivalent to ADD dest,1 or SUB dest,1. Because CF is affected, INC and DEC can be used for address arithmetic if CF needs to be preserved. Faster and shorter than ADD or SUB

NEG (NEGate)

Purpose:
    Compute 2's complement negation; subtracts operand from 0 and replaces it with result
Syntax:
    NEG dest
Semantics:
    dest <- 0 - dest
Flags:
    O...SZAP.   modified for result
   C   set if dest <> 0
    .DIT.....   Unchanged
Operands:
        reg   mem

Conversion Instructions: CBW, CWD, CWDE [386], CDQ [386]
MOVSX [386], MOVZX [386]

CBW (Convert Byte to Word)

Purpose:
    Sign extend a signed byte to a word
Syntax:
    CBW
Semantics:
    AX <- AL sign extended
Flags
    ODITSZAPC   unchanged

CWD (Convert Word to Doubleword)

Purpose:
    Sign extend a signed word to a signed doubleword
Syntax:
    CWD
Semantics:
    DX:AX <- AX sign extended
Flags:
    ODITSZAPC   unchanged

CWDE (Convert Word to Doubleword Extended) [386]

Purpose:
    Sign extend a signed word to a signed doubleword
Syntax:
    CWDE
Semantics:
    eax <- AX sign extended
Flags:
    ODITSZAPC   unchanged

CDQ (Convert Doubleword to Quadword)[386]

Purpose:
    Sign extend a signed word to a signed doubleword
Syntax:
    CDQ
Semantics:
    edx:eax <- eax sign extended
Flags:
    ODITSZAPC   unchanged

Notes:
1. These are one-operand instructions-registers affected cannot be changed
2. AH or DX are destroyed
3. To convert an unsigned operand just zero out (zero-extend) high-order places.
4. MOVSX and MOVZX are generalizations of these instructions

MOVSX (MOV with Sign-Extend)/MOVZX (Mov with Zero-Extend) [386]

Purpose:
    Copy a byte to word or dword with sign or zero-extension; convert a word to dword with sign or zero extension
Syntax:
    MOVSX reg, source
    MOVZX reg, source
Semantics:
    reg <- source sign or zero extended
Flags:
    ODITSZAPC   unchanged
Operands:
    reg, mem8   reg, mem16   reg, reg8   reg,reg16
Notes:
    Dest operand must be a general purpose register

Multiplication and Division

MUL (unsigned MULtiply)
IMUL (signed Integer MULtiply)

Purpose:
    Multiply two unsigned integers (MUL) or signed integers (IMUL)
Syntax:
    MUL source
    IMUL source
Semantics:
    AX <-- AL * source8   OR:
    DX:AX <-- AX * source16   OR:
    edx:eax <- eax * source32
Flags:
    O.......C   modified for product
    ....SZAP.   undefined
    .DIT.....   unchanged
Operands:
    reg   mem
Notes on MUL and IMUL:
1. Overflow and Carry flags indicate if low-order product valid
2. 8088 does not have multiply by immediate value

IMUL 2 & 3 operand

Purpose:
    Multiply register by immediate value or register, for small products only. High order part of product is truncated.
Syntax:
    imul dest, imm (80186)
    imul dest, source, imm (80186)
    imul dest, source (80386)
Semantics:
    dest <- dest * imm
    dest <- source * imm
    dest <- dest * source
Flags:
    O.......C   clear if product is valid (no high-order part)
    ....SZAP.   undefined
    .DIT.....   unchanged
Operands:

reg16, imm	reg32, imm
reg16, reg16, imm	reg32, reg32, imm
reg16, mem16, imm	reg32, mem32, imm
reg16, reg16	reg16, mem16
reg32, reg32	reg32, mem32

DIV (unsigned DIVision)
IDIV (signed Integer DIVision)

Syntax:
    DIV source
    IDIV source
Semantics:
    AL <-- quotient of AX / source8
    AH <-- remainder of AX / source8
OR:
    AX <-- quotient of DX:AX / source16
    DX <-- remainder of DX:AX / source16

    Processor generates INT 0 exception if quotient too large or attempt to divide by 0
Flags:
    O...SZAPC   undefined
    .DIT.....   unchanged
Operands:
    reg mem
Notes:
1. Divide overflow or attempt to divide by zero results in processor generating INT 0; default action is to halt program and return control to DOS.
2. Be careful with negative numbers-remainder is negative

BCD Arithmetic: DAA, DAS, AAA, AAS, AAM, AAD
(See Intel references for more detailed semantics)

DAA (Decimal Adjust for Addition)

Purpose:
    After a packed BCD addition in AL, adjusts the sum in AL and CF
Syntax:
    DAA
Semantics:
    Adjust AL to record correct packed BCD results.
    Force carry bit if result > 99
Flags:
    ....SZAPC   modified
    O........   undefined
    .DIT.....   unchanged

DAS (Decimal Adjust for Subtraction)

Purpose:
    After a packed BCD subtraction in AL, adjusts AL and CF
Syntax:
    DAS
Semantics:
    Adjust AL to record correct packed BCD results.
    Force carry bit if there is a borrow out of AL.
Flags:
    ....SZAPC   modified
    O........   undefined
    .DIT.....   unchanged
Notes:
    See references for more information on this instruction.

AAA (Ascii Adjust after Addition)

Purpose:
    After an unpacked BCD addition in AL, adjusts AL and reflects carry in AH and CF
Syntax:
    AAA
Semantics:
    If AL>9 then AL <- AL-10; AH <- AH+1; CF <- 1; AF<- 1
            else CF <- 0; AF <- 0
Flags:
    ......A.C   modified
    O...SZ.P.   undefined
    .DIT.....   unchanged

AAS (Ascii Adjust after Subtraction)

Purpose:
    After an unpacked BCD subtraction in AL, adjusts AL and reflects borrow in AH
    and CF
 Syntax:
    AAS
Semantics:
    If AL<0 then AL <- AL+10; AH <- AH-1; CF <- 1; AF <- 1
            else CF <- 0; AF <- 0
Flags:
    ......A.C   modified
    O...SZ.P.   undefined
    .DIT.....   unchanged

AAM (Ascii Adjust after Multiplication)

Purpose:
    Adjust AL and AH to BCD after an unpacked BCD multiplication of AL.
Syntax:
    AAM   (Undocumented: AAM imm)
Semantics:
    AH <- AL / 10; AL <- AL mod 10 (AH <- AL / imm; AL <- AL mod imm)
Flags:
    ....SZ.P.   modified
    O.....A.C   undefined
    .DIT.....   unchanged
Notes:
    The undocumented AAM imm is supported by some assemblers. If imm=16, can be used
    to unpack packed BCD

AAD (Ascii Adjust before Division)

Purpose:
Converts two unpacked BCD digits in AL and AH into one binary number in AX so that a DIV instruction can be correctly executed.

Syntax:
```
AAD      (Undocumented: AAD imm)
```
Semantics:
```
AX <- 10 * AH + AL
```
Flags:

....SZ.P.	modified
O.....A.C	undefined
.DIT.....	unchanged

Notes:
1. Unlike other BCD instructions, AAD is executed BEFORE the division is performed.
2. The undocumented AAD imm is supported by some assemblers. If imm=16, can be used to pack unpacked BCD

Logical Operations: AND, OR, NOT, XOR, TEST

AND

Purpose:
Compute logical (bitwise) AND of two operands

Syntax:
```
AND dest,source
```
Semantics:
```
dest <- dest AND source
```
flags:

....SZ.P	modified for result
O......C	cleared
......A.	undefined
.DIT....	unchanged

operands:

reg,reg	mem,reg	reg,mem
reg,immed	mem,immed	

Notes:
Useful for clearing individual bits--this process is called masking

TEST

Purpose:
Compute logical AND and set flags but do not store result.
Always used in conjunction with conditional jumps.

Syntax:
```
TEST op1,op2
```
Semantics:
compute (op1 AND op2) and modify flags

Flags: Same as AND

Operands: Same as AND

Notes:
TEST is similar to CMP

OR

Purpose:
Compute logical (bitwise) OR of two operands

Syntax:
```
OR dest,source
```
Semantics:
```
dest <- dest OR source
```
flags:

....SZ.P	modified for result
O......C	cleared
......A.	undefined
.DIT....	unchanged

operands:

reg,reg	mem,reg	reg,mem
reg,immed	mem,immed	

Notes:
Useful for setting individual bits

XOR (eXclusive OR)

Purpose:
    Compute bitwise XOR. Often used to complement part of a word.
Syntax:
    XOR source,dest
Semantics:
    dest <- dest XOR source
flags:

....SZ.P	modified for result
O......C	cleared
......A.	undefined
.DIT....	unchanged

operands:

reg,reg	mem,reg	reg,mem
reg,immed	mem,immed	

Notes:
1. To complement part of an operand, create a mask that has a 1 in each position to be complemented and a 0 in each position to be left unchanged
2. Can be used to clear a register: XOR AX,AX. Same effect and speed as subtraction: SUB AX,AX. either is faster than MOV AX,0

NOT

Purpose:
    Used to logically negate or 1's complement an operand
Syntax:
    NOT dest
Semantics:
    dest <- NOT dest
Flags:
    ODITSZAPC unchanged
operands:

reg	mem

Notes:
    See NEG for two's complement negation

Shifts: SAL, SHL, SAR, SHR, SHLD, SHRD

SHL (SHift logical Left)
SAL (SHift Arithmetic Left)

Purpose:
    Shift bits in destination operand one or more bits to the left. There is no difference between logical and arithmetic left shifts.
Syntax:
    1. SHL dest, 1   2. SHL dest, CL   3. SHL dest, imm (186)
Semantics:
    1. bits N..1 of dest <- bits N-1..0 of dest; bit 0 <- 0
    2. bits N..CL of dest <- bits N-CL..0 of dest; bits CL-1..0 <- 0
    3. bits N..imm of dest <- bits N-imm..0 of dest; bits imm-1..0 <- 0
Flags:

....SZ.PC	modified for result
O.....A..	undefined
.DIT.....	unchanged

Operands:

reg,1	reg,CL	reg, imm (186)
mem,1	mem,CL	mem, imm (186)

Notes:
1. Carry flag gets last bit shifted out of operand; other flags have same interpretation as a MULtiply instruction. Overflow flag is defined only for SHL dest,1 and SAL dest,1.
2. Can be used to implement quick multiplication by powers of 2.

SHR (SHift logical Right)

Purpose:
Shift bits in destination operand one or more bits to the right.
Syntax:
1. SHR dest, 1 2. SHR dest, CL 3. SHR dest, imm (186)
Semantics:
1. bits N-1..0 of dest <- bits N..1 of dest; bit N <- 0
2. bits N-CL..0 of dest <- bits N..CL of dest; bits N..N-CL+1 <- 0
3. bits N-imm..0 of dest <- bits N..imm of dest; bits N..N-imm+1 <- 0
Flags:

....SZ.PC	modified for result
O.....A..	undefined
.DIT.....	unchanged

Operands:
Same as SHL
Notes:
1. Carry flag gets last bit shifted out of register; other flags have same interpretation as a DIVide instruction. OF is defined only for SHR dest,1.
1. Can be used to implement quick unsigned division by powers of 2.

SAR (Shift Arithmetic Right)

Purpose:
Shifts bits to the right in the destination operand. Leftmost bits are filled from the most significant bit of the operand, thus preserving the sign of the operand.
Syntax:
1. SAR dest, 1 2. SAR dest, CL 3. SAR dest, imm (186)
Semantics:
1. bits N-1..0 of dest <- bits N..1 of dest; bit N <- bit N
2. bits N-CL..0 of dest <- bits N..CL of dest; bits N..N-CL+1 <- bit N
2. bits N-imm..0 of dest <- bits N..imm of dest; bits N..N-imm+1 <-bit N
Flags:

....SZ.PC	modified for result
O.....	cleared
......A..	undefined
.DIT.....	unchanged

Operands:
Same as SHL
Notes:
1. Carry flag gets last bit shifted out of register; other flags have same interpretation as a DIVide instruction. Overflow flag is always cleared.
2. Can be used to implement quick signed division by powers of 2 with appropriate precautions to ensure that operands are within ranges that produce correct results.

SHLD/SHRD (Shift Left/Right Double) [386]

Purpose:
Copies bits from source operand into dest operand, shifting dest before copy.
Syntax:
SHLD dest, source, imm SHRD dest, source, imm
Semantics:
SHLD: dest <- dest:source shifted R by imm, source is unaltered
SHRD: dest <- source:dest shifted L by imm, source is unaltered
Flags:

....SZ.P.	modified for result
.....C	copy of last bit that was shifted
O.....	cleared
......A..	undefined
.DIT.....	unchanged

Operands:

reg16/mem16, reg16, imm8	reg16/mem16, reg16, CL
reg32/mem32, reg32, imm8	reg32/mem32, reg32, CL

Rotates: ROL, ROR, RCL, RCR

ROL (ROtate Left)

Purpose:
Rotate 8, 16 or 32 bits in destination operand one or more bits to the left.
Syntax:
1. ROL dest, 1 2. ROL dest, CL 3. ROL dest, imm (186)
Semantics:
1. bits N..1 of dest <- bits N-1..0; bit 0 <- bit N
2. bits N..CL of dest <- bits N-CL..0; bits CL-1..0 <- bits N..N-CL+1
3. bits N..imm of dest <- bits N-imm..0; bits CL-1..0 <- bits N..N-imm+1
Flags:
.....C gets last bit rotated out
O..... undefined
.DITSZAP. unchanged
Operands:
reg,1 reg,CL mem,1 mem,CL reg,imm (186) mem/imm (186)

ROR (ROtate Right)

Purpose:
Rotate 8, 16, or 32 bits in destination operand one or more bits to right.
Syntax:
1. ROR dest, 1 2. ROR dest, CL 3. ROR dest,imm (186)
Semantics:
1. bits N-1..0 of dest <- bits N..1 ; bit N <- bit 0
2. bits N-CL..0 of dest <- bits N..CL; bits N..N-CL+1 <- bits CL-1..0
3. bits N-imm..0 of dest <- bits N..imm; bits N..N-imm+1 <- bits imm-1..0
Flags:
.....C gets last bit rotated out
O..... undefined
.DITSZAP. unchanged
Operands:
reg,1 reg,CL mem,1 mem,CL reg,imm (186) mem/imm (186)

RCL (ROtate with Carry Left)

Purpose:
Rotate 9, 17, or 33 bits in destination operand one or more bits to the left.
Carry flag is considered the leftmost bit of the operand.
Syntax:
1. RCL dest, 1 2. RCL dest, CL 3. RCL dest,imm (186)
Semantics:
1. bits N..1 of dest <- bits N-1..0; bit 0 <- CF; CF <- bit N
2. bits N..CL of dest <- bits N-CL..0; bit CL-1 <- CF, bits CL-2..0 <- bits N..N-CL+2; CF <- bit CL-1
3. bits N..imm of dest <- bits N-imm..0; bit imm-1 <- CF, bits imm-2..0 <- bits N..N-imm+2; CF <- bit imm-1
Flags:
.....C gets last bit rotated out
O..... undefined
.DITSZAP. unchanged
Operands:
reg,1 reg,CL mem,1 mem,CL reg,imm (186) mem/imm (186)

RCR (ROtate with Carry Right)

Purpose:
Rotate 9, 17, or 33 bits in destination operand one or more bits to right.
Carry flag is considered the leftmost bit of the operand.
Syntax:
1. RCR dest, 1 2. RCR dest, CL 3. RCR dest, imm (186)
Semantics:
1. bits N-1..0 of dest <- bits N..1; bit N <- CF; CF <- bit 0
2. bits N-CL..0 of dest <- bits N..CL; bit N-CL+1 <- CF; bits N..N-CL+2<- bits CL-2..0; CF <- bit CL-1
2. bits N-imm..0 of dest <- bits N..imm; bit N-imm+1 <- CF; bits N..N-imm+2<- bits imm-2..0; CF <- bit imm-1
Flags:
.....C gets last bit rotated out
O..... undefined
.DITSZAP. unchanged
Operands:
reg,1 reg,CL mem,1 mem,CL reg,imm (186) mem/imm (186)

Bit Scan: BSF (Bit Scan Forward) [386]
BSR (Bit Scan Reverse) [386]

Purpose:
Scan source operand for the first set bit and leave position in dest operand. BSF scans from 0 up and BSR from MSB down. ZF if set if source is 0 (no 1 bit was found), clear otherwise

Syntax:
1. BSF dest,source 2. BSR dest, source

Flags:

.....Z...	Set if source=0, clear otherwise
ODITS.APC	unchanged

Semantics:
dest <- bit position of first set bit found, unchanged if source is 0

Operands:

reg16, mem16	reg16, reg16
reg32, reg32	reg32, mem32

Notes:
1. Destination operand must be a register
2. No 8-bit operands for either dest or source

Bit Test[386]: BT (Bit Test), BTS (Bit Test and Set)
BTR (Bit Test and Reset), BTC (Bit Test and Complement)

Purpose:
Test a specified bit by copying it to CF. BTS will also set the bit after it is copied; BTR will clear (reset) the bit after it is copied, and BTR will complement the bit after it is copied.

Syntax:
BT dest, source BTS dest, source BTR dest, source BTC dest, source

Semantics:
CF <- bit in dest specified by source; dest modified by BTS, BTR, BTC as specified above

Flags:

.........C	gets copy of bit from dest
ODITSZAP.	unchanged

Operands:

reg, reg	mem, reg
reg, imm8	reg, imm8

Subroutines: CALL, RET, RETF
CALL

Purpose:
Call a subroutine and stack return address.

Syntax:
CALL dest

Semantics:

1.	SP <- SP -2	esp <- esp-4
2.	[SS:SP] <- IP	[ss:esp] <- eip
3.	IP <- dest	eip <- dest

Note that a FAR CALL will also stack CS then load CS with the new segment address.

Flags:

ODITSZAPC	unchanged

Operands:

label	reg	mem

Note:
Syntax of far call depends on assembler.

RET (RETurn) and RETF (RETurn Far)

Purpose:

Return from a subroutine, popping IP or CS:IP from the stack. RETF forces a far return. An immediate value as an operand adjusts the stack pointer by that amount after popping the return address.

Syntax:

1.	RET	2. RETF
3.	RET immed	4. RETF immed

Semantics:

1.	IP <- [SS:SP]	eip <- [ss:esp]
	SP <- SP + 2	esp <- esp + 4
2.	IP <- [SS:SP]	eip <- [ss:esp]
	CS <- [SS:SP+2]	cs <- [ss:esp+4]
	SP <- SP + 4	esp <- esp + 6
3.	IP <- [SS:SP]	eip <- [ss:esp}
	SP <- SP + 2 + immed	esp <- esp + 4 + immed
4.	IP <- [SS:SP]	eip <- [ss:esp}
	CS <- [SS:SP+2]	cs <- [ss:esp+4]
	SP <- SP + 4 + immed	esp <- esp + 6 + immed

Notes:

Even though data can only be stacked as words or dword, the stack adjustment operand is specified in bytes. Therefore it must always be EVEN.

JMP (JuMP)

Purpose:

Perform an unconditional jump.

Syntax:

JMP dest

Semantics

IP <- dest

Flags:

ODITSZAPC unchanged

Operands:

index reg mem label

Notes:

1. Can write SHORT JMP, NEAR JMP or FAR JMP, but most assemblers accept just JMP and figure out from context which type of jump is appropriate

Conditional Jumps

Notes on Conditional Jumps:

1. Before the 80386 Conditionals are all short jumps (-128 to +127 bytes). While this saves space, it is often a bit of a nuisance and sometimes requires grouping a conditional with a JMP. Starting with the 80286, conditionals can be near (16-bit signed offset) in 16-bit code segments, or near (32-bit signed offset) in 32-bit code segments.

2. Following is a summary of conditionals:

Type of Test	Mnemonics
Zero:	JE, JZ, JNE, JNZ
Unsigned arithmetic:	JB, JNAE, JBE, JNA, JA, JNBE, JAE, JNB
Signed Arithmetic:	JL, JNGE, JLE, JNG, JG, JNLE, JGE, JNL
Overflow:	JO, JNO
Carry:	JC, JNC
Sign:	JS, JNS
Parity	JP, JPE, JNP, JPO
Test for CX = 0:	JCXZ

3. The sense of a comparison conditional such as JAE (jump if above or equal) folowing a CMP can be understood by placing the equivalent relational operator between dest and source. For example:

```
cmp eax, ebx
jae target1
```

The jump is taken if eax >= ebx

Syntax & Semantics of Conditional Jumps

All have same syntax and semantics for varying conditions.

Syntax:

    Jxxx dest

Semantics:

    IP <- dest if condition tested is true

Flags:

    ODITSZAPC   unchanged

Operands:

    Short Label   Near Label (386)

Zero Tests (JE, JZ, JNE, JNZ)

Purpose:

    Jump if zero flag set or clear.

JE (Jump if Equal)
JZ (Jump if Zero)

Semantics:

    IP <- dest if ZF = 1

JNE (Jump if Not Equal)
JZ (Jump if Not Zero)

Semantics:

    IP <- dest if ZF = 0

Signed Conditionals
JL, JNGE, JLE, JNG, JG, JNLE, JGE, JNL

Purpose:

    Used after signed comparisons

JL   (Jump if Less)
JNGE   (Jump if not Greater or Equal)

Semantics:

    IP <- dest if SF <> OF

JLE   (Jump if Less or Equal)
JNG   (Jump if not Greater)

Semantics:

    IP <- dest if ZF = 1 or SF <> OF

JG   (Jump if Greater)
JNLE   (Jump if not Less or Equal)

Semantics:

    IP <- dest if ZF = 0 or SF = OF

JGE   (Jump if Greater or Equal)
JNL   (Jump if not Less)

Semantics:

    IP <- dest if SF = OF

Note that semantics are expressed in terms of flags but English mnemonics make a lot more sense

Unsigned Conditionals
JB, JNAE, JBE, JNA, JA, JNBE, JAE, JNB

Purpose:

    Used after unsigned comparisons

JB   (Jump if Below)
JNAE   (Jump if not Above or Equal)

Semantics:

    IP <- dest if CF = 1

JBE   (Jump if Below or Equal)
JNA   (Jump if not Above)

Semantics:

    IP <- dest if CF = 1 or ZF = 1

JA   (Jump if Above)
JNBE   (Jump if not Below or Equal)

Semantics:

    IP <- dest if CF = 0 and ZF = 0

JAE   (Jump if Above or Equal)
JNA   (Jump if not Below)

Semantics:

    IP <- dest if CF = 0

Individual Flag Tests
(JO, JNO, JC, JNC, JS, JNS, JP, JPE, JNP, JPO)

Purpose:
    Test overflow, carry, sign, or parity flag and jump if test true.

JO   (Jump if Overflow)
JNO   (Jump if no Overflow)
Semantics:
    JO: IP <- dest if OF = 1
    JNO: IP <- dest if OF = 0

JC   (Jump if Carry)
JNC   (Jump if no Carry)
Semantics:
    JC: IP <- dest if CF = 1
    JNC: IP <- dest if CF = 0

JS   (Jump if Sign)
JNS   (Jump if no Sign)
Semantics:
    JS: IP <- dest if SF = 1
    JNS: IP <- dest if SF = 0
Remember that SF = 1 if result is negative

JP   (Jump if Parity)
JPE   (Jump if Parity Even)
Semantics:
    IP <- dest if PF = 1

JNP   (Jump if Parity)
JPO   (Jump if Parity Even)
Semantics:
    IP <- dest if PF = 0
Remember that PF = 1 on even parity

JCXZ (Jump if CX Zero)
JECXZ (Jump if ECX Zero) [386]

Purpose:
    This jump instruction inspects CX or ecx, not the flags, but does not modify the flags
Semantics:
    IP <- dest if CX = 0
Flags:
    ODITSZAPC   unchanged
Note:
    This is mostly of interest as a test before LOOP instructions

Conditional SETcc [386]

Purpose:
    The conditional SETs take a single byte dest operand and write a 1 to the operand if condition is true, 0 if false. Conditions are same as conditional jmp (Jcc) instructions:
        seta, setae, setb, setbe, setc, sete, setg, setge, setl, setle,setna, setnae, setnb, setnbe, setnc, setne, setng, setnge, setnl, setnle, setno, setnp, setns, setnz, seto, setp, setpe, setpo, sets, setz
    Can be used to avoid pipeline stalls cause by conditional jumps.
Syntax:
    SETcc dest
Semantics:
    dest <- 1 if condition true, 0 otherwise (See Jcc)
Flags:
    ODITSZAPC   unchanged
Operands:
    reg8   mem8

LOOP

Purpose:
Loop while CX (ecx) is non-zero, decrementing CX (ecx) for each iteration. Like a FOR loop in a high-level language

Syntax:
LOOP dest LOOPD dest (386) LOOPW dest (386)

Semantics:
(1) CX <- CX-1 (no effect on flags) LOOPD:ecx <- ecx-1 (no effect on flags)
(2) if CX != 0 then IP <-- dest LOOPD:if ecx != 0 then IP <-- dest

Flags:
ODITSZAPC unchanged

Operands:
Short label only (-128 to +127 bytes)

Notes:
1. Note CX or ecx decremented before test
2. Often JCXZ (jecxz) used before a LOOP because of predecrement semantics.
3. 386+ LOOP uses ecx, use LOOPW to specify CX in 32-bit mode or LOOPD to specify CX in 16-bit mode

Conditional Loops: LOOPE/LOOPDE [386] and LOOPNE/LOOPDNE [386]

Purpose:
Loop while CX > 0 and ZF clear or set. These loops have TWO termination tests.
LOOPE (LOOP if Equal)
LOOPZ (LOOP if Zero)

Syntax:

LOOPE dest	LOOPEW dest (386)	LOOPED dest (386)
LOOPZ dest	LOOPZW dest (386)	LOOPZD dest (386)

Semantics:
(1) CX <- CX - 1 (no effect on flags)
(2) if CX > 0 AND ZF = 1 then IP <-- dest

Flags:
ODITSZAPC unchanged

Operands:
Short label only (-128 to +127 bytes)

Note:
1. Since test is an AND, loop will terminate when either ZF = 0 OR CX = 0. This loop is typically used when "searching for a mismatch"
2. 386+ LOOPE uses ecx, use W suffix to specify CX in 32-bit mode or D suffix to specify ecx in 16-bit mode

LOOPNE (LOOP if Not Equal)
LOOPNZ (LOOP if Not Zero)

Syntax:

LOOPNE dest	LOOPNEW dest (386)	LOOPNED dest (386)
LOOPNZ dest	LOOPNZW dest (386)	LOOPNZD dest (386)

Semantics:
(1) CX <-- CX - 1 (flags unchanged)
(2) if CX > 0 AND ZF = 0 then IP <- dest

Flags:
ODITSZAPC unchanged

Operands:
Short label only (-128 to +127 bytes)

Note:
1. This loop is typically used when "searching for a match"
2. 386+ LOOPNE uses ecx, use W suffix to specify CX in 32-bit mode or D suffix to specify ecx in 16-bit mode

XLAT/XLATB (X transLATE)

Purpose:
Table lookup instruction that allows easy performance of tasks such as ASCII to EBCDIC conversion; binary to hex conversion. XLAT and XLATB are the same instruction

Syntax:
```
XLAT    or    XLATB
```
Semantics:
```
AL <- DS:[BX+AL]          al <- ds:[ebx+al] (386)
```
Flags:
ODITSZAPC unchanged

Notes:
1. Consider the following:
```
TABLE DB '0123456789ABCDEF'
...
MOV BX,TABLE
AND AL,0FH
XLAT
```
AL gets the ASCII char corresponding to the lower nibble.
2. XLAT does not know the table size so if value in AX is larger than the table XLAT will cheerfully retrieve whatever garbage happens to lie above the table.

String Instructions MOVS,CMPS,LODS,STOS,SCAS

These instructions are often used with a REP prefix and a count in CX or ecx. If a source operand is in memory (MOVS,CMPS,LODS) then DS:[SI] or ds:[esi] is the source operand. If a destination operand is in memory (MOVS,CMPS,STOS,SCAS) then ES:[DI] or es:[edi] is the destination operand. The index register(s) are auto-incremented or auto-decremented by the size of the operand according to the state of DF (0 = increment). Also INS and OUTS.

LODSB (LOad String Byte)
LODSW (LOad String Word)
LODSD (LOad String Dword) [386]

Purpose:
Copy byte or word from DS:[SI] into AL or AX, or ds:[esi] in eax and update SI or esi accordingly. Can be used with repeat prefixes but there is little point in doing so as each repetition will overwrite the data in the accumulator.

Syntax:
```
LODSB         LODSW             LODSD (386)
```
Semantics:
```
(1)     AL <- DS:[SI]  (byte)
        AX <- DS:[SI]  (word)
        eax <- ds:[esi] (dword)
(2)     If DF = 0 then SI <- SI + 1 else SI <- SI - 1 (byte)
        If DF = 0 then SI <- SI + 2 else SI <- SI - 2 (word)
        If DF = 0 then esi <- esi + 4 else esi <- esi - 2 (word)
```
Flags:
ODITSZAPC unchanged

STOSB (STOre String Byte)
STOSW (STOre String Word)
STOSD (STOre String Dword) [386]

Purpose:
Copy byte or word from AL or AX to ES:[DI], or eax to es:[edi] and update DI or edi accordingly. Can be used with REP prefix to fill a block of memory with a single value.

Syntax:
```
STOSB         STOSW             STOSD (386)
```
Semantics:
```
(1)     ES:[DI] <- AL  (byte)
        ES:[DI] <- AX  (word)
        es:[edi] <- eax (dword)
(2)     If DF = 0 then SI <- SI + 1 else SI <- SI - 1 (byte)
        If DF = 0 then SI <- SI + 2 else SI <- SI - 2 (word)
        If DF = 0 then esi <- esi + 4 else esi <- esi - 4 (dword)
```
Flags:
ODITSZAPC unchanged

Notes:
LODS followed by STOS allows inspection of the byte/word/dword whereas MOVS does not. This can be useful in situations such as copying a string where it necessary to know when the zero terminating byte has been copied.

MOVSB (MOVe String Byte)
MOVSW (MOVe String Word)
MOVSD (MOVe String Dword)[386]

Purpose:
    To copy a byte or word from DS:SI to ES:DI. (386 ds:esi to es:edi) Can be used with repeat prefix to copy entire strings.
Syntax:
    MOVSB   MOVSW   MOVSD (386)
Semantics:
```
(1)      ES:[DI] <- DS:[SI] (byte or word)
(2)      If DF = 0 then SI <- SI+1 and DI <- DI+1
                   else SI <- SI-1 and DI <- DI-1  (byte)
         If DF = 0 then SI <- SI+2 and DI <- DI+2
                   else SI <- SI-2 and DI <- DI-2  (word)
         If DF = 0 then esi <- esi+4 and edi <- edi+4
                   else esi <- esi-4 and edi <- edi-4 (dword)
```
Flags:
    ODITSZAPC unchanged
Notes:
    To move a string set up DS:SI (ds:esi) to point to source and ES:DI (es:edi) to point to destination. Then clear the direction flag, load the count CX and execute MOVSB, MOVSW, or MOVSD with a REP prefix.

CMPSB (COMpare String Byte)
CMPSW (COMpare String Word)
CMPSD (COMpare String Dword) [386]

Purpose:
    Similar to CMP instruction, but operates memory-to-memory and is usually used with REPNE or REPE to locate first match or mismatch in a string.
Syntax:
    CMPSB   CMPSW   CMPSD (386)
Semantics:
```
(1)      Compute DS:[SI] - ES:[DI] and set flags
(2)      If DF = 0 then SI <- SI+1 and DI <- DI+1
                   else SI <- SI-1 and DI <- DI-1  (byte)
         If DF = 0 then SI <- SI+2 and DI <- DI+2
                   else SI <- SI-2 and DI <- DI-2  (word)
         If DF = 0 then esi <- esi+4 and edi <- edi+4
                   else esi <- esi-4 and edi <- edi-4 (dword)
```
Flags:
```
    O...SZAPC      modified for results of comparison
    .DIT.....      unchanged
```
Notes:
1. Normally used with REPE and REPNE prefixes. When used with REPE will find position of first mismatch; when used REPNE will find position of first match.
2. When looking for matches or mismatches bear in mind that SI and DI are pointing to the location AFTER the one you want because of the post-increment nature of the instruction. Therefore DECREMENT THEM if you want the correct locations. (Reverse this if DF = 1!)

SCASB (SCAn String Byte)
SCASW (SCAn String Word)
SCASD (SCAn String DWord)

Purpose:
    Similar to CMP instruction with AL, AX or eax, but is usually used with REPNE or REPE to locate a particular character or value in a string.
Syntax:
    SCASB   SCASW   CMPSD (386)
Semantics:
```
(1)      Compute AL - ES:[DI] and set flags  (byte)
         Compute AX - ES:[DI] and set flags  (word)
         Compute eax- es:[edi] and set flags  (Dword)
(2)      If DF = 0 then DI <- DI+1 else DI <- DI-1  (byte)
         If DF = 0 then DI <- DI+2 else DI <- DI-2  (word)
         If DF = 0 then DI <- DI+4 else DI <- DI-4  (dword)
```
Flags:
```
    O...SZAPC      modified for results of comparison
    .DIT.....      unchanged
```
Notes:
1. Normally used with REPE and REPNE prefixes. When used with REPE will find position of first mismatch; when used REPNE will find position of first match.

2. When looking for matches or mismatches bear in mind that DI is pointing to the location AFTER the one you want because of the post-increment nature of the instruction. Therefore DECREMENT DI if you want the correct location. (Reverse this if DF = 1!)

REP (REPeat)

Purpose:
Used before a string instruction to force an unconditional repetition of the instruction for the number of times specified in CX
Syntax:
REP string instruction
Semantics:
execute string instruction the number of times specified in CX. CX <- CX - 1 AFTER each repetition. Terminates when CX = 0.
Flags:
As per string instruction

Note: used with MOVSB, MOVSW, STOSB, STOSW, INS and OUTS instructions.

REPE (REPeat while Equal)
REPZ (REPeat while Zero)

Purpose:
Used before a string instruction to force repetition of the instruction for the number of times specified in CX but only while Zero flag is set.
Syntax:
REP string instruction
Semantics:
execute string instruction the number of times specified in CX. CX <- CX - 1 AFTER each repetition.
Terminates when CX = 0 OR ZF = 1
Flags:
As per string instruction

Note: used with CMPSB, CMPSW, SCASB, SCASW instructions to search for position of first mismatch.

REPNE (REPeat while Not Equal)
REPNZ (REPeat while Not Zero)

Purpose:
Used before a string instruction to force repetition of the instruction for the number of times specified in CX but only while Zero flag is clear.

Syntax:
REPNE string instruction
REPNZ string instruction
Semantics:
execute string instruction the number of times specified in CX. CX <- CX - 1 AFTER each repetition.
Terminates when CX = 0 OR ZF = 0
Flags:
As per string instruction
Note: used with CMPSB, CMPSW, SCASB, SCASW instructions to search for position of first match.

Interrupts: INT, INTO, IRET

INT (INTerrupt)

Purpose:
    Transfer control to specified interrupt handler.
Syntax:
    INT immed
Semantics:
    1. 16-bit: PUSH Flags, CS, IP; 32-bit: PUSH Eflags, CS and eip
    2. (Real Mode) CS:IP <- Vector located at 0000:(4 * immed)
        (Protected Mode) Vector descriptor at [IDTR]+8*immed
Flags:
    OD..SZAPC   unchanged
    ..IT.....   cleared
Operands:
    Immediate value 0 .. 255.
Notes:
1. Interrupts are like a FAR CALL.
2. Vector table occupies the first 1K bytes of memory.
3. Interrupt flag is cleared, disabling further interrupts until flag is reset with a STI instruction.
4. Int 3 (Breakpoint) generates a different opcode from any other interrupt. This is a one-byte opcode that allows a break point to be set at any instruction.
5. Unlike CALL, flags register is preserved by INT instruction. Note that an interrupt handler COULD destroy other registers. All handlers preserve any registers they use; therefore ALL programs must have a small amount of space on the stack (at least 128 bytes) for use by interrupt handlers.

INTO (INTerrupt on Overflow)

Purpose:
    Transfer control to INT 4 handler if an overflow has occurred.
Syntax:
    INTO
Semantics:
    1. 16-bit: PUSH Flags, CS, IP; 32-bit: PUSH Eflags, CS and eip
    2. (Real Mode) CS:IP <- Vector located at 0000:0010
        (Protected Mode) Vector descriptor at [IDTR]+0020
Flags:
    OD..SZAPC   unchanged
    ..IT.....   cleared
Notes:
1. This is a conditional instruction similar to JO.
2. Default overflow handler is just an IRET instruction.

IRET (Interrupt RETurn)
IRETD (32-bit Interrupt RETurn) [386]

Purpose:
    Interrupt Return is like a far Return.
Syntax:
    IRET   IRETD
Semantics:
    1. IRET: POP IP, CS and Flags in order
    2. IRETD: POP eip, CS and EFlags in order
Flags:
    Restored from stack
    32-bit semantics are very complex in protected mode.

Flag Control
CLI, STI, CLC, STC, CMC, CLD, STD

Purpose:
    Set or clear Direction, Interrupt, or Carry flags (CMC is Complement Carry Flag)

CLC (CLear Carry flag)
Syntax: CLC
Semantics: CF <- 0

STC (SeT Carry flag)
Syntax: STC
Semantics: CF <- 1

CMC (CoMplement Carry flag)
Syntax: CMC
Semantics: CF <- CF-1

CLD (CLear Direction flag)
Syntax: CLD
Semantics: DF <- 0; String ops performed in ascending order

STD (SeT Direction flag)
Syntax: STD
Semantics: DF <- 1; String ops performed in descending order

CLI (CLear Interrupt enable flag)
Syntax: CLI
CLI Semantics: IF <- 0; Interrupts are DISABLED
Note: This is a privileged instruction in protected mode and will cause a general protection fault if attempted without sufficient privilege.

STI (SeT Interrupt enable flag)
Syntax: STI
Semantics: IF <- 1; Interrupts are ENABLED
Note: This is a privileged instruction in protected mode and will cause a general protection fault if attempted without sufficient privilege.

Miscellaneous
NOP, BSWAP,BOUND, ENTER, LEAVE

NOP (No OPeration)

Purpose:
Performs no operation; simply consumes time. Used for several reasons: (1) to word-align or paragraph-align instructions; (2) as a filler in ML programs in case additional instructions need to be added; (3) to construct delay loops; (4) for timing benchmarks; (5) to fine-tune operation of the prefetch queue.
Instruction actually assembled is XCHG AX,AX
Syntax:
NOP

BSWAP (Byte Swap) [486]

Purpose:
Reverse order of bytes in operand. Used for endian translations.
Syntax:
BSWAP dest
Semantics:
Reverses order of bytes in dest
Flags:
ODITSZAPC unchanged
Operands:
reg32

CPUID CPU identification (Pentium+)

BOUND

Purpose:
Test an array address against bounds of the array. INT 5 exception raised if test fails.
Syntax:
BOUND source, limit
Semantics:
Array index in source register is checked against upper and lower bounds in memory source. The first word located at "limit" is the lower boundary and the word at "limit+2" is the upper array bound. INT 5 raised if the source value is less than or higher than the bounds
Flags:
ODITSZAPC unchanged
Operands:
reg16, mem32 reg32, mem64

ENTER [186]

Purpose:
Set up stack frame
Syntax
ENTER imm, imm
Semantics:
See Intel references

LEAVE [186]

Purpose:
    Remove stack frame
Syntax
    LEAVE
Semantics:
    See Intel references

Process Synchronization
XADD, CMPXCHG, CMPXCHG8B

These rather strange instructions are used to provide atomic (uninterruptible) instructions for use in implementing mutual exclusion objects such as semaphores and mutexes. The LOCK prefix is required to guarantee uninterruptible operation.

XADD Exchange and add [486]

Purpose:
    Takes 2 operands, exchanges their contents and writes the sum of the two operands into the first operand
Syntax:
    XADD dest1, dest2
Semantics:
    dest2 <- dest1; dest1 <- dest2 + dest1
Flags:
    O...SZAPC   modified for sum
    .DIT.....   unchanged
Operands:
    reg, mem   reg,reg

Note:
    This bizarre instruction can be used to write the world's fastest Fibonacci calculation:

```
        mov ecx, N
        Sub edx, edx
        mov eax, 1
    FibLoop:
        xadd eax, edx
        loop FibLoop
```

CMPXCHG Compare and Exchange [486]

Purpose:
    Compares the accumulator (8-32 bits) with dest. If equal the dest is loaded with src, otherwise the accumulator is loaded with dest.
Syntax:
    CMPXCHG dest, src
Semantics:

```
    if (dest=accum)
        dest <- src
    else
        accum <- dest
```

Flags:
    O...SZAPC   modified for difference
    .DIT.....   unchanged
Operands:
    mem,reg

CMPXCHG8B Compare and Exchange 8 bytes [Pentium]

Purpose:
    This instruction compares the 64 bit value in edx:eax with the memory value. If they are equal, the Pentium stores ecx:ebx into the memory location, otherwise it loads edx:eax with the memory location. This instruction sets the zero flag according to the result. It does not affect any other flags.
Syntax:
    CMPXCHG8B mem64
Semantics:

```
    if (mem64=accum)
        mem64 <- ecx:ebx
    else
        edx:eax <- mem64
```

Flags:
    ODITS.APC   unchanged
    .....Z...   Result of comparison
Operands:
    mem64

Bus and Device Synchronization
ESC, HLT, WAIT, LOCK

ESC (ESCape)

Purpose:
Signals the arithmetic coprocessor that it is to execute the next instruction.
Syntax:
Not used in assembly language. The assembler provides this instruction when 80x87 instructions are encountered in the program.

HLT (HaLT)

Purpose:
Halts the processor until one of three events occurs: (1) an interrupt; (2) a hardware reset or (3) DMA operation concludes. Normally only used in a program to synchronize with a hardware device.
Syntax:
HLT

WAIT

Purpose:
Monitors the TEST pin on the 8086 (name changed to BUSY in later processors). If TEST = 0 there is no effect. If TEST = 1 then the processor halts until TEST = 0.

The TEST pin is usually connected to the TEST pin of the coprocessor; thus the 8086 halts until the coprocessor has finished its current operation. Note that some operations on the coprocessor can take over 1000 clocks.
Syntax:
WAIT
Note:
Often inserted automatically by assemblers following a coprocessor instruction.

LOCK prefix

Purpose:
Sets the lock pin on the processor and disables external bus master, thus preventing access to memory or other resources. Used to lock memory during critical operations.
Syntax:
LOCK:instruction

Systems Programming Instructions

These instruction are not discussed in this document.

ARPL	Adjust RPL field of selector (286+)
CLTS	Clear task switched flag in CR0 (286+)
CPUID	Return information on processor model and capabilities (Pentium)
INVD	Invalidate data cache (486+)
INVLPG	Invalidate TLB entry (486+)
LAR	Load access rights byte (286+)
LGDT	Load global descriptor table register (286+)
LIDT	Load interrupt descriptor table register (286+)
LLDT	Load local descriptor table register (286+)
LMSW	Load machine status word (286+)
LSL	Load segment limit (286+)
LTR	Load task register (286+)
RDMSR	Read Model-Specific Register (Pentium)
RSM	Resume from system management mode (Pentium+)
SGDT	Store global descriptor table register (286+)
SIDT	Store interrupt descriptor table register (286+)
SLDT	Store local descriptor table register (286+)
SMSW	Store machine status word (286+)
VERR	Verify a segment for reading (286+)
VERW	Verify a segment for writing (286+)
WBINVD	Write-back and invalidate data cache (486+)
WRMSR	Write to model specific register (PENTIUM+)