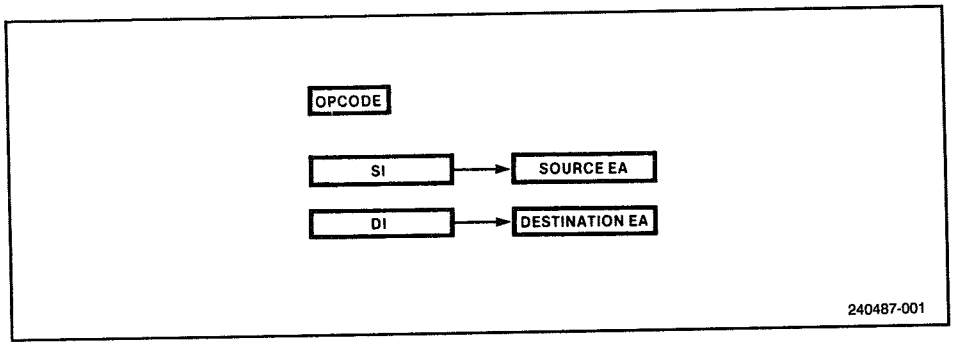

Figure 6-25. String Operand Addressing

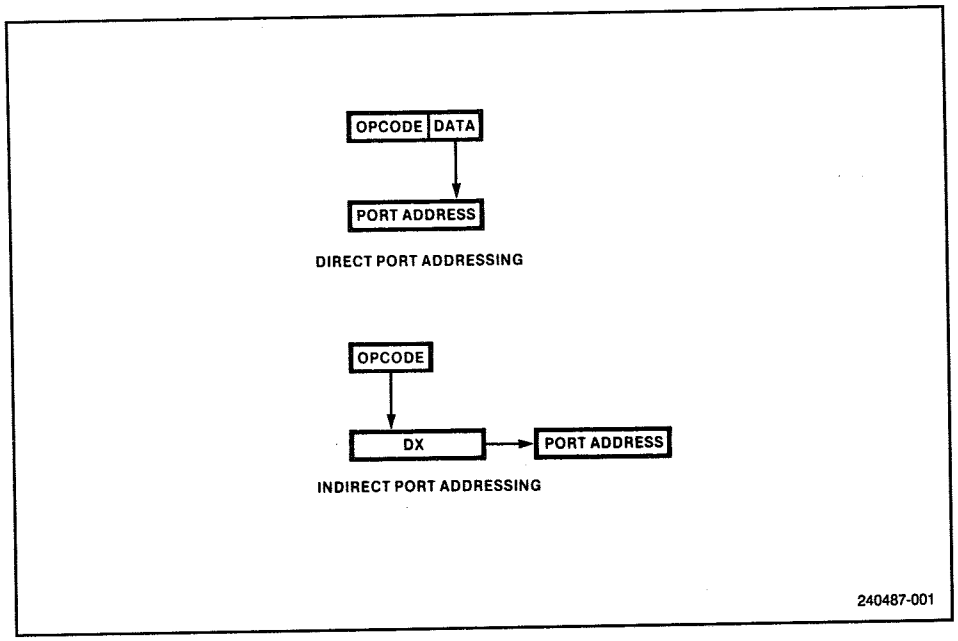

Figure 6-26. I/O Port Addressing

allows fixed access to ports numbers 0–255. Indirect I/O port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and can range from 0 to 65,535. By previously adjusting the content of register DX, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value of DX.

INSTRUCTION SET SUMMARY

The following paragraphs, and tables, provide detailed information for the 8086/8088 instruction set. Table 6-12, 6-13 and 6-14 explain the symbols that are used in Table 6-16, the instruction set reference data table. Machine language instruction encoding and decoding information is provided in the paragraphs immediately following the instruction set summary.

Instruction timings are presented as the number of clock periods required to execute a particular form of the instruction (register-to-register, immediate-to-memory, etc.). If the system is running with a 5 MHz maximum clock, the maximum clock period is 200 ns; at 8 MHz, the clock period is 125 ns. When memory operands are used, "+EA" indicates a variable number of additional clock periods needed to calculate the operand's effective address. Table 6-15 lists all effective address calculation times.

The timings given for control transfer instructions include any additional clocks required to reinitialize the instruction queue as well as the time required to fetch the target instructions. For instructions executing on an 8086, four clocks should be added for each instruction reference to a word operand located at an odd memory address to reflect any additional operand bus cycles required. Also, for instructions executing on an 8088, four clocks should be added to each instruction reference to a 16-bit memory operand. This includes stack operations. The required number of data references is listed for each instruction in Table 6-16 to aid in this calculation.

All of the instruction times given are of the form "n," where "n" is the number of clocks required for the 8086 to execute the given instruction. The number of clocks required for the 8088 will be n for 8-bit operations and n + (4* transfers) for 16-bit operations.

For instructions which repeat a specified number of times, the value n consists of two parts in the relation "x + y/rep," where x is the initial number of clocks required to start the instruction, and y is the number of clocks corresponding to the number of iterations specified. For 16-bit repeated instructions on the 8088 when the expression "(4 * transfers)" has to be added to n, it should be added to the y part of the expression before it is multiplied by the number of repetitions.

Several additional factors can alter the actual execution time from the figures shown in Table 6-16. The time provided assumes that the instruction has already been prefetched and that it is waiting in the instruction queue. This assumption is valid under most, but not all, operating conditions. A series of fast executing (fewer than two clocks per opcode byte) instructions can drain the queue and increase execution time. Execution time is also slightly effected by the interaction of the EU and BIU when memory operands must be read or written. If the EU needs access to memory, it may have to wait for up to one clock if the BIU has already started an instruction fetch bus cycle. (The EU can detect the need for a memory operand and post a bus request far enough in advance of its need for this operand to avoid waiting a full 4-clock bus cycle.) If the queue is full the EU does not have to wait because the BIU is idle. (This assumes the BIU can obtain the bus on demand and no other processors are competing for the bus.)

With typical instruction mixes, the time actually required to execute a sequence of instructions will be within 5-10% of the sum of the individual timings provided in Table 6-16. Cases can be constructed, however, in which execution time may be much higher than the sum of the figures provided in the table. The execution time for a given sequence of instructions is always repeatable, assuming comparable external conditions (interrupts, coprocessor activity, etc.). If the execution time for a given series of instructions must be determined exactly, the instructions should be run on an execution vehicle such as the iSBC 88/25 or 86/30 board.

Table 6-12. Key to Instruction Coding Formats

Identifier	Used In	Explanation
destination	data transfer, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location or immediate value that is used in the operation, but is not altered by the instruction.
source-table	XLAT	Name of memory translation table addressed by register BX.
target	JMP, CALL	A label to which control is to be transferred directly, or a register or memory location whose *content* is the address of the location to which control is to be transferred indirectly.
short-label	cond. transfer, iteration control	A label to which control is to be conditionally transferred; must lie within −128 to +127 bytes of the first byte of the next instruction.
accumulator	IN, OUT	Register AX for word transfers, AL for bytes.
port	IN, OUT	An I/O port number; specified as an immediate value of 0-255, or register DX (which contains port number in range 0-64K).
source-string	string ops.	Name of a string in memory that is addressed by register SI; used only to identify string as byte or word and specify segment override, if any. This string is used in the operation, but is not altered.
dest-string	string ops.	Name of string in memory that is addressed by register DI; used only to identify string as byte or word. This string receives (is replaced by) the result of the operation.
count	shifts, rotates	Specifies number of bits to shift or rotate; written as immediate value 1 or register CL (which contains the count in the range 0-255).
interrupt-type	INT	Immediate value of 0-255 identifying interrupt pointer number.
optional-pop-value	RET	Number of bytes (0-64K, ordinarily an even number) to discard from stack.
external-opcode	ESC	Immediate value (0-63) that is encoded in the instruction for use by an external processor.

Table 6-13. Key to Flag Effects

Identifier	Explanation
(blank)	not altered
0	cleared to 0
1	set to 1
X	set or cleared according to result
U	undefined – contains no reliable value
R	restored from previously-saved value

Table 6-14. Key to Operand Types

Identifier	Explanation
(no operands)	No operands are written
register	An 8- or 16-bit general register
reg 16	A 16-bit general register
seg-reg	A segment register
accumulator	Register AX or AL
immediate	A constant in the range 0-FFFFH
immed8	A constant in the range 0-FFH
memory	An 8- or 16-bit memory location*
mem8	An 8-bit memory location*
mem16	A 16-bit memory location*
source-table	Name of 256-byte translate table
source-string	Name of string addressed by register SI
dest-string	Name of string addressed by register DI
DX	Register DX
short-label	A label within −128 to +127 bytes of the end of the instruction
near-label	A label in current code segment
far-label	A label in another code segment
near-proc	A procedure in current code segment
far-proc	A procedure in another code segment
memptr16	A word containing the offset of the location in the current code segment to which control is to be transferred*
memptr32	A doubleword containing the offset and the segment base address of the location in another code segment to which control is to be transferred*
regptr16	A 16-bit general register containing the offset of the location in the current code segment to which control is to be transferred
repeat	A string instruction repeat prefix

*Any addressing mode – direct, register indirect, based, indexed, or based indexed – may be used.

Table 6-15. Effective Address Calculation Time

EA Components		Clocks*
Displacement Only		6
Base or Index Only (BX, BP, SI, DI)		5
Displacement		9
+		
Base or Index	(BX, BP, SI, DI)	
Base	BP + DI, BX + SI	7
+	BP + SI, BX + DI	8
Index		
Displacement	BP + DI + DISP	11
+	BX + SI + DISP	
Base		
+	BP + SI + DISP	12
Index	BX + DI + DISP	

*Add 2 clocks for segment override

Table 6-16. Instruction Set Reference Data

AAA — **AAA** (no operands) ASCII adjust for addition — Flags O D I T S Z A P C: U U U X U X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	8	–	1	AAA

AAD — **AAD** (no operands) ASCII adjust for division — Flags O D I T S Z A P C: U X X U X U

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	60	–	2	AAD

AAM — **AAM** (no operands) ASCII adjust for multiply — Flags O D I T S Z A P C: U X X U X U

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	83	–	2	AAM

AAS — **AAS** (no operands) ASCII adjust for subtraction — Flags O D I T S Z A P C: U U U X U X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	8	–	1	AAS

ADC — **ADC** destination, source — Add with carry — Flags O D I T S Z A P C: X X X X X X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	–	2	ADC AX, SI
register, memory	9 + EA	1	2-4	ADC CX, BETA [SI]
memory, register	16 + EA	2	2-4	ADC ALPHA [BX] [SI], DI
register, immediate	4	–	3-4	ADC BX, 256
memory, immediate	17 + EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	–	2-3	ADC AL, 5

ADD — **ADD** destination, source — Addition — Flags O D I T S Z A P C: X X X X X X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	–	2	ADD CX, DX
register, memory	9 + EA	1	2-4	ADD DI,[BX], ALPHA
memory, register	16 + EA	2	2-4	ADD TEMP, CL
register, immediate	4	–	3-4	ADD CL, 2
memory, immediate	17 + EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	–	2-3	ADD AX, 200

AND — **AND** destination, source — Logical and — Flags O D I T S Z A P C: O X X U X O

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	–	2	AND AL, BL
register, memory	9 + EA	1	2-4	AND CX, FLAB WORD
memory, register	16 + EA	2	2-4	AND ASCII [DI], AL
register, immediate	4	–	3-4	AND CX, 0F0H
memory, immediate	17 + EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	–	2-3	AND AX, 01010000B

CALL — **CALL** target — Call a procedure — Flags O D I T S Z A P C: X X X X X X

Operands	Clocks	Transfers*	Bytes	Coding Example
near-proc	19	1	3	CALL NEAR_PROC
far-proc	28	2	5	CALL FOR_PROC
memptr 16	21 + EA	2	2-4	CALL PROC_TABLE [SI]
regptr 16	16	1	2	CALL AX
memptr 32	37 + EA	4	2-4	CALL [BX].TASK [SI]

CBW — **CBW** (no operands) Convert byte to word — Flags O D I T S Z A P C: U U U X U X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	–	1	CBW

CLC	CLC (no operands) Clear carry flag			Flags O D I T S Z A P C O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	CLC

CLD	CLD (no operands) Clear direction flag			Flags O D I T S Z A P C O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	CLD

CLI	CLI (no operands) Clear interrupt flag			Flags O D I T S Z A P C O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	CLI

CMC	CMC (no operands) Complement carry flag			Flags O D I T S Z A P C X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	CMC

CMP	CMP destination, source Compare destination to source			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	–	2	CMP BX, CX
register, memory	9 + EA	1	2-4	CMP DH, ALPHA
memory, register	9 + EA	1	2-4	CMB [BP + 2], SI
register, immediate	4 + EA	–	3-4	CMP BL, 02H
memory, immediate	10 + EA	1	3-6	CMB [BX].RADAR [DI], 3420H
accumulator, immediate	4	–	2-3	CMP AL, 00010000B

CMPS	CMPS des-string, source-string Compare string			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	22	2	1	CMPS BUSS1, BUFF2
(repeat) dest-string, source-string	9 + 22/rep	2/rep	1	REPE CMPS ID, KEY

CWD	CWD (no operands) Convert word to doubleword			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	5	–	1	CWD

DAA	DAA (no operands) Decimal adjust for addition			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	–	1	DAA

DAS	DAS (no operands) Decimal adjust for subtraction			Flags O D I T S Z A P C U X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	–	1	DAS

DEC	DEC destination Decrement by 1			Flags O D I T S Z A P C X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 16	3	–	1	DEC AX
reg 8	3	–	2	DEC AL
memory	15 + EA	2	2-4	DEC ARRAY [SI]

DIV	DIV source Division, unsigned			Flags O D I T S Z A P C U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 8	80-90	–	2	DIV CL
reg 16	144-162	–	2	DIV BX
mem 8	89-96 + EA	1	2-4	DIV ALPHA
mem 16	150-168 + EA	1	2-4	DIV TABLE [SI]

ENTER	ENTER Procedure entry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
locals, level	L = 0 L = 1 L > 1 (22 + 16(n – 1))	–	4	ENTER 28, 3

ESC	ESC external-opcode, source Escape			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immediate, memory immediate, register	8 + EA 2	1 –	2-4 2	ESC 6,ARRAY [SI] ESC 20, AL

HLT	HLT (no operands) Halt			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	HLT

IDIV	IDIV source Integer division			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 8 reg 16 mem 8 mem 16	101-112 165-184 107-118 + EA 171-190 + EA	– – 1 1	2 2 2-4 2-4	IDIV BL IDIV CX IDIV DIVISOR_BYTE [SI] IDIV [BX].DIVISOR_WORD

IMUL	IMUL source Integer multiplication			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 8 reg 16 mem 8 mem 16	80-98 128-154 86-104 + EA 134-160 + EA	– – 1 1	2 2 2-4 2-4	IMUL CL IMUL BX IMUL RATE_BYTE IMUL RATE_WORD [BP] [DI]

IN	IN accumulator, port Input byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, immed 8 accumulator, DX	10 8	1 1	2 1	IN AL, 0FFEAH IN AX, DX

INC	INC destination Increment by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 16 reg 8 memory	3 3 15 + EA	– – 2	1 2 2-4	INC CX INC BL INC ALPHA [DI] [BX]

INT	INT Interrupt-type Interrupt			Flags O D I T S Z A P C O O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed 8 (type = 3) immed 8 (type 3)	52 52	5 5	1 2	INT 3 INT 67

INTR†	INTR (external maskable interrupt) Interrupt if INTER and IF = 1			Flags O D I T S Z A P C O O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	61	7	N/A	N/A

INTO	INTO (no operands) Interrupt if overflow			Flags O D I T S Z A P C O O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	53 or 4	5	1	INTO

IRET	IRET (no operands) Interrupt Return			Flags O D I T S Z A P C R R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	32	3	1	IRET

JA/JNBE	JA/JNBE short-label Jump if above/Jump if not below nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JA ABOVE

JAE/JNB	JAE/JNB short-label Jump if above or equal/Jump if not below			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JAE ABOVE_EQUAL

JB/JNAE	JB/JNAE Jump if below/Jump if not above nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JB BELOW

JBE/JNA	JBE/JNA short-label Jump if below or equal/Jump if not above			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JNA NOT_ABOVE

JC	JC short-label Jump if carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JC CARRY_SET

JCXZ	JCXZ short-label Jump if CX is zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JCXZ COUNT_DONE

JE/JZ	JE/JZ short-label Jump if equal/Jump if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JZ ZERO

JG/JNLE	JG/JNLE short-label Jump if greater/Jump if not less nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JG GREATER

JGE/JNL	JGE/JNL short-label Jump if greater or equal/Jump if not less			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JGE GREATER_EQUAL

JL/JNGE	JL/JNGE short-label Jump if less/Jump if not greater nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JL LESS

JLE/JNG	JLE/JNG short-label Jump if less or equal/jump if not greater			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JNG NOT_GREATER

JMP	JMP target Jump			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	15	–	2	JMP SHORT
near-label	15	–	3	JMP WITHIN_SEGMENT
far-label	15	–	5	JMP FAR_LABEL
memptr 16	18 + EA	1	2-4	JMP [BX].TARGET
regptr 16	11	–	2	JMP CX
memptr 32	24 + EA	2	2-4	JMP OTHER_SEG [SI]

JNC	JNC short-label Jump if not carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JNC NOT_CARRY

JNE/JNZ	JNE/JNZ short-label Jump if not equal/Jump if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JNE NOT_EQUAL

JNO	JNO short-label Jump if not overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JNO NO_OVERFLOW

JNP/JPO	JNP/JPO short-label Jump if not parity/Jump if parity odd			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JPO ODD_PARITY

JNS	JNS short-label Jump if not sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JNS POSITIVE

JO	JO short-label Jump if overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JO SIGNED_OVERFLOW

JP/JPE	JP/JPE short-label Jump if parity/Jump if parity even			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JPE EVEN_PARITY

JS	JS short-label Jump if sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	–	2	JS NEGATIVE

LAHF	LAHF (no operands) Load AH from flags			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	–	1	LAHF

LDS	LDS destination, source Load pointer using DS			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 16, mem 16	16 + EA	2	2-4	LDS SI, DATA, SEG [DI]

LEA	LEA destination, source Load effective address			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 16, mem 16	2 + EA	–	2-4	LEA BX, [BP] [DI]

LES	LES destination, source Load pointer using ES			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg 16, mem 32	16 + EA	2	2-4	LES DI,[BX], TXT_BUFF

LOCK	LOCK (no operands) Lock bus			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	LOCK XCHG FLAG, AL

LODS	LODS source-string Load string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-string (repeat) source-string	12 9 + 13/rep	1 1/rep	1 1	LODS CUSTOMER_NAME REP LODS NAME

LOOP	LOOP short-label Loop			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	17	–	2	LOOP AGAIN

LOOPE/LOOPZ	LOOPE/LOOPZ short-label Loop if equal/Loop if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	–	2	LOOPE AGAIN

LOOPNE/LOOPNZ	LOOPNE/LOOPNZ short-label Loop if not equal/Loop if not zero			Flags O D I T S Z A P C O O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	19 or 5	–	2	LOOPNE AGAIN

NMI†	NMI (external nonmaskable interrupt) Interrupt if NMI = 1			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

MOV

MOV destination, source
Move

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
memory, accumulator	10	1	3	MOV ARRAY [SI], AL
accumulator, memory	10	1	3	MOV AZ, TEMP_RESULT
register, register	2	–	2	MOV AZ, CX
register, memory	8 + EA	1	2-4	MOV BP, STACK_TOP
memory, register	9 + EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	–	2-3	MOV CL, 2
memory, immediate	10 + EA	1	3-6	MOV MASK [BX] [SI], 2CH
seg-reg, reg 16	2	–	2	MOV ES, CX
seg-reg, mem 16	8 + EA	1	2-4	MOV DS, SEGMENT_BASE
reg 16, seg-reg	2	–	2	MOV BP, SS
memory, seg-reg	9 + EA	1	2-4	MOV [BX], SEG_SAVE, CS

MOVS

MOVS dest-string, source-string
Mov string

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
dest-string, source-string	18	2	1	MOVS LINE EDIT_DATA
(repeat) dest-string, source-string	9 + 17/rep	2/rep	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW

MOVSB/MOVSW (no operands)
Move string (byte/word)

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	18	2	1	MOVSB
(repeat)(no operands)	9 + 17/rep	2/rep	1	REP MOVSW

MUL

MUL source
Multiplication, unsigned

Flags O D I T S Z A P C
X U U U U X

Operands	Clocks	Transfers*	Bytes	Coding Example
reg 8	70-77	–	2	MUL BL
reg 16	118-133	–	2	MUL CX
mem 8	76-83 + EA	1	2-4	MUL MONTH [SI]
mem 16	124-139 + EA	1	2-4	MUL BAUD_RATE

NEG

NEG destination
Negate

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
register	3	–	2	NEG AL
memory	16 + EA	2	2-4	NEG MULTIPLIER
*0 if destination is 0				

NOP

NOP (no operands)
No operation

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	3	–	1	NOP

NOT

NOT destination
Logical not

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
register	3	–	2	NOT AX
memory	16 + EA	2	2-4	NOT CHARACTER

OR

OR destination, source
Logical inclusive or

Flags O D I T S Z A P C
O X X U X O

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	–	2	OR AL, BL
register, memory	9 + EA	1	2-4	OR DX, PORT_ID [DI]
memory, register	16 + EA	2	2-4	OR FLAG_BYTE, CL
accumulator, immediate	4	–	2-3	OR AL, 011011008
register, immediate	4	–	3-4	OR CX, 01H
memory, immediate	17 + EA	2	3-6	OR [BX], CMD_WORD, 0CFH

OUT

OUT port, accumulator
Output byte or word

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
immed 8, accumulator	10	–	2	OUT 44, AX
DX, accumulator	8	1	1	OUT DX, AL

POP

POP destination
Pop word off stack

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
register	8	1	1	POP DX
seg-reg (CS illegal)	8	1	1	POP DS
memory	17 + EA	2	2-4	POP PARAMETER

POPF

POPF (no operands)
Pop all registers

Flags O D I T S Z A P C
R R R R R R R R R

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	8	1	1	POPF

PUSH	**PUSH** source Push word onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	11	1	1	PUSH SI
seg-reg (CS legal)	10	1	1	PUSH ES
memory	16 + EA	2	2-4	PUSH RETURN_CODE [SI]

PUSHF	**PUSHF** (no operands) Push flags onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	10	1	1	PUSHF

RCL	**RCL** destination, count Rotate left through carry			Flags O D I T S Z A P C X C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	RCL CX, 1
register, CL	8 + 4/bit	–	2	RCL AL, CL
memory, 1	15 + EA	2	2-4	RCL ALPHA, 1
memory CL	20 + 4/bit + EA	2	2-4	RCL [BP], PARM, CL

RCR	**RCR** destination, count Rotate right through carry			Flags O D I T S Z A P C X C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	RCR BX, 1
register, CL	8 + 4/bit	–	2	RCR BL, CL
memory, 1	15 + EA	2	2-4	RCR [BX], STATUS, 1
memory CL	20 + 4/bit + EA	2	2-4	RCR ARRAY, [DI], CL

REP	**REP** (no operands) Repeat string operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	REP MOVS DEST, SRCE

REPE/REPZ	**REPE/REPZ** (no operands) Repeat string operation while equal/while zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	REPE CMPS DATA, KEY

REPNE/REPNZ	**REPNE/REPNZ** (no operands) Repeat string operation while not equal/ not zero			Flags O D I T S Z A P C U UUXUX
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	REPNE SCAS INPUT_LINE

RET	**RET** optional-pop-value Return from procedure			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(intra-segment, no pop)	16	1	1	RET
(intra-segment, pop)	20	1	3	RET 4
(inter-segment, no pop)	26	2	1	RET
(inter-segment, pop)	25	2	3	RET 2

ROL	**ROL** destination, count Rotate left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	ROL BX, 1
register, CL	8 + 4/bit	–	2	ROL DI, CL
memory, 1	15 + EA	2	2-4	ROL FLAG_BYTE [DI], 1
memory CL	20 + 4/bit + EA	2	2-4	ROL ALPHA, CL

ROR	**ROR** destination, count Rotate right			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	ROR BX, 1
register, CL	8 + 4/bit	–	2	ROR BX, CL
memory, 1	15 + EA	2	2-4	ROR PORT_STATUS, 1
memory CL	20 + 4/bit + EA	2	2-4	ROR CMD_WORD, CL

SAHF	**SAHF** (no operands) Store AH into flags			Flags O D I T S Z A P C R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	–	1	SAHF

SAL/SHL	SAL/SHL destination Shift arithmetic left/Shift logical left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	SAL AL, 1
register, CL	8 + 4/bit	–	2	SAL DI, CL
memory, 1	15 + EA	2	2-4	SAL [BX], OVERDRAW, 1
memory, CL	20 + 4/bit + EA	2	2-4	SAL STORE_COUNT, CL

SAR	SAR destination, source Shift arithmetic right			Flags O D I T S Z A P C X X X U X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	SAR DX, 1
register, CL	8 + 4/bit	–	2	SAR DI, CL
memory, 1	15 + EA	2	2-4	SAR N_BLOCKS, 1
memory, CL	20 + 4/bit + EA	2	2-4	SAR N_BLOCKS, CL

SBB	SBB destination, source Subtract with borrow			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	–	2	SBB BX, CX
register, memory	9 + EA	1	2-4	SBB DI, [BX], PAYMENT
memory, register	16 + EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	–	2-3	SBB AX, 2
register, immediate	4	–	3-4	SBB CL, 1
memory, immediate	17 + EA	2	3-6	SBB COUNT, [SI], 10

SCAS	SCAS dest-string Scan string			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	15	1	1	SCAS INPUT_LINE
(repeat) dest-string	9 + 15/rep	1/rep	1	REPNE SCAS BUFFER

SEGMENT†	SEGMENT override, prefix Override to specified segment			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	MOV SS: PARAMETER AX

SHR	SHR destination, count Shift logical right			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	–	2	SHR SI, 1
register, CL	8 + 4/bit	–	2	SHR SI, CL
memory, 1	15 + EA	2	2-4	SHR ID BYTE [SI] [BX],1
memory, CL	20 + 4/bit + EA	2	2-4	SHR INPUT WORD, CL

SINGLE STEP††	SINGLE STEP (Trap flag interrupt) Interrupt if TF = 1			Flags O D I T S Z A P C O O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

STC	STC (no operands) Set carry flag			Flags O D I T S Z A P C C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	STC

STD	STD (no operands) Set direction flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	STD

STI	STI (no operands) Set interrupt enable flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	–	1	STI

STOS	STOS dest-string Store byte or word string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	11	1	1	STOS PRINT_LINE
(repeat) dest-string	9 + 10/rep	1/rep	1	REP STOS DISPLAY

SUB	SUB destination, source Subtraction			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	–	2	SUB CX, BX
register, memory	9 + EA	1	2-4	SUB DX, MATH_TOTAL [SI]
memory, register	16 + EA	2	2-4	SUB [BP+2], CL
accumulator, immediate	4	–	2-3	SUB AL, 10
register, immediate	4	–	3-4	SUB SI, 5280
memory, immediate	17 + EA	2	3-6	SUB [BP], BALANCE, 1000

TEST	TEST destination, source Test or non-destructive logical and			Flags O D I T S Z A P C O X X U X O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	–	2	TEST SI, DI
register, memory	9 + EA	1	2-4	TEST SI, END_COUNT
accumulator, immediate	4	–	2-3	TEST AL, 001000008
register, immediate	5	–	3-4	TEST BX, 0CC4H
memory, immediate	11 + EA	–	3-6	TEST RETURN_COUNT, 01H

WAIT	WAIT (no operands) Wait while TEST pin not asserted			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4 + 5	–	1	WAIT

XCHG	XCHG destination, source Exchange			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, reg 16	3	–	1	XCHG AX, BX
memory, register	17 + EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	–	2	XCHG AL, BL

XLAT	XLAT source-table Translate			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-table	11	1	1	XLAT ASCII_TAB

XOR	XOR destination, source Logical exclusive or			Flags O D I T S Z A P C O X X U X O
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	–	2	XOR CX, BX
register, memory	9 + EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16 + EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	–	2-3	XOR AL, 010000108
register, immediate	4	–	2-4	XOR SI, 00C2H
memory, immediate	17 + EA	2	3-6	XOR RETURN_CODE 0D2H

*For the 8086 add four clocks for each 16-bit word transfer with an odd address. For the 8088 add four clocks for each 16-bit word transfer.

†INTR is not an instruction, it is included in Table 6-16 only for timing information.

†NMI is not an instruction, it is included in Table 6-16 only for timing information.

†ASM-86 incorporates the segment override prefix into the operand specification and not as a separate instruction. SEGMENT is included in Table 6-16 only for timing information.

††SINGLE STEP is not an instruction, it is included in Table 6-16 only for timing information.

MACHINE INSTRUCTION ENCODING AND DECODING

Machine instruction encoding and decoding is primarily the concern of the programmer. It is presented here for the hardware designer since such encoding and decoding directly affects bus activity. As an example of the encoding and decoding process, consider writing a MOV instruction in ASM-86 in the form:

MOV destination,source

This will cause the assembler to generate 1 of 28 possible forms of the MOV machine instruction. A programmer rarely needs to know the details of machine instruction formats or encoding. An exception may occur during debugging when it may be necessary to monitor instructions fetched on the bus, read unformatted memory dumps, etc. This section provides the information necessary to translate or decode an 8086 or 8088 machine instruction.

To pack instructions into memory as densely as possible, the 8086 and 8088 CPUs utilize an efficient coding technique. Machine instructions vary from one to six bytes in length. One-byte instructions, which generally operate on single registers or flags, are simple to identify; the keys to decoding longer instructions are in the first two bytes. The format of these bytes can vary, but most instructions follow the format shown in Figure 6-27.

The first six bits of a multibyte instruction generally contain an opcode that identifies the basic instruction type: ADD, XOR, etc. The following bit, called the D field, generally specifies the "direction" of the operation: 1 = the REG field in the second byte identifies the destination operand, 0 = the REG field identifies the source operand. The W field distinguishes between byte and word operations: 0 = byte, 1 = word.

"r/m" Field Bit Assignments

r/m	Operand Address
000	(BX) + (SI) + DISP
001	(BX) + (DI) + DISP
010	(BP) + (SI) + DISP
011	(BP) + (DI) + DISP
100	(SI) + DISP
101	(DI) + DISP
110	(BP) + DISP
111	(BX) + DISP

DISP follows 2nd byte of instruction (before data if required).
*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

AAA — ASCII ADJUST FOR ADDITION

Operation

```
if ((AL) & 0FH) >9 or (AF) = 1 then
   (AL) ← (AL) + 6
   (AH) ← (AH) + 1
   (AF) ← 1
(CF) ← (AF)
(AL) ← (AL) & 0FH
```

Flags Affected

AF, CF
OF, PF, XF, ZF undefined

Description

AA (ASCII Adjust for Addition) changes the contents of register AL valid unpacked decimal number; the high-order half-byte is zeroed. updates AF and CF; the content of OF, PF, SF and ZF is undefined fo ing execution of AAA.

Encoding

0 0 1 1 0 1 1 1

AAA Operands	Clocks	Transfers	Bytes	AAA Coding Exampl
(no operands)	8	–	1	AAA

AAD — ASCII ADJUST FOR DIVISION

Operation

(AL) ← (AH) *0AH + (AL)
(AH) ← 0

Flags Affected

PF, SF, ZF
AF, CF, OF undefined

Description

AAD (ASCII Adjust for Division) modifies the numerator in AL *before* dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. AAD updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAD.

Encoding

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

AAD Operands	Clocks	Transfers	Bytes	AAD Coding Example
(no operands)	60	–	2	AAD

AAM — ASCII ADJUST FOR MULTIPLY

Operation

(AH) ← (AL)/0AH
(AL) ← (AL)%0AH

Flags Affected

PF, SF, ZF
AF, CF, OF undefined

Description

AAM (ASCII Adjust for Multiply) corrects the result of a previc cation of two valid unpacked decimal operands. A valid 2-dig decimal number is derived from the content of AH and AL and to AH and AL. The high-order half-bytes of the multiplied ope have been 0H for AAM to produce a correct result. AAM upd and ZF; the content of AF, CF and OF is undefined following e AAM.

Encoding

1 1 0 1 0 0 0 0	0 0 0 0 1 0 1 0

AAM Operands	Clocks	Transfers	Bytes	AAM Coding Exar
(no operands)	83	–	2	AAM

AAS — ASCII ADJUST FOR SUBTRACTION

Operation

```
if ((AL) & 0FH) >9 or (AF) = 1 then
  (AL) ← (AL) −6
  (AH) ← (AH) −1
  (AF) ← 1(CF) ← (AF)
(AL) ← (AL) & 0FH
```

Flags Affected

AF, CFOF, PF, SF, ZF undefined

Description

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAS.

Encoding

0 0 1 1 1 1 1 1

AAS Operands	Clocks	Transfers	Bytes	AAS Coding Example
(no operands)	8	–	1	AAS

ADC — ADD WITH CARRY

Operation

```
if (CF) = 1 then (DEST) ← (LSRC)
  + (RSRC) + 1
else (DEST) ← (LSRC) + (RSRC)
```

Flags Affected

AF, CF, OF, PF, SF, ZF

Description

ADC *destination, source*

ADC (Add with Carry) sums the operands, which may be byte
adds one if CF is set and replaces the destination operand with
Both operands may be signed or unsigned binary numbers (see
DAA). ADC updates AF, CF, OF, PF, SF and ZF. Since ADC i
a carry from a previous operation, it can be used to write rout
numbers longer than 16 bits.

Encoding

Memory or Register Operand with Register Operand

0 0 0 1 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Immediate Operand to Memory or Register Operand

1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s:\

LSRC = EA, RSRC = data, DEST = EA

Immediate Operand to Accumulator

0 0 0 1 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

ADC Operands	Clocks	Transfers	Bytes	ADC Coding E
register, register	3	–	2	ADC AX, SI
register, memory	9 + EA	1	2-4	ADD DX, BETA
memory, register	16 + EA	2	2-4	ADC ALPHA [B
register, immediate	4	–	3-4	ADC BX, 256
memory, immediate	17 + EA	2	3-6	ADC GAMMA, 3
accumulator, immediate	4	–	2-3	ADC AL, 5

ADD — ADDITION

Operation (DEST) ← (LSRC) + (RSRC)

Flags Affected AF, CF, OF, PF, SF, ZF

Description **ADD** *destination, source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF and ZF.

Encoding

Memory or Register Operand with Register Operand

0 0 0 0 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Immediate Operand to Memory or Register Operand

1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s:w = 01

LSRC = EA, RSRC = data, DEST = EA

Immediate Operand to Accumulator

0 0 0 0 0 1 0 w	data	data if w = 1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

ADD Operands	Clocks	Transfers	Bytes	ADD Coding Example
register, register	3	–	2	ADD CX, DX
register, memory	9 + EA	1	2-4	ADD DI, [BX],ALPHA
memory, register	16 + EA	2	2-4	ADD TEMP, CL
register, immediate	4	–	3-4	ADD CL, 2
memory, immediate	17 + EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	–	2-3	ADD AX, 200

AND — AND LOGICAL

Operation (DEST) ← (LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

Flags Affected CF, OF, PF, SF, ZFAF undefined

Description **AND** *destination, source*

AND performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.

Encoding

Memory or Register Operand with Register Operand

0 0 1 0 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Immediate Operand to Memory or Register Operand

1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1

LSRC = EA, RSRC = data, DEST = EA

Immediate Operand to Accumulator

0 0 1 0 0 1 0 w	data	data if w = 1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

AND Operands	Clocks	Transfers	Bytes	AND Coding Example
register, register	3	–	2	AND AL, BL
register, memory	9 + EA	1	2-4	AND CX, FLAG WORD
memory, register	16 + EA	2	2-4	AND ASCII [DI],AL
register, immediate	4	–	3-4	AND CX, 0F0H
memory, immediate	17 + EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	–	2-3	AND AX, 01010000B

CALL — CALL PROCEDURE

Operation

```
if Inter-Segment then
  (SP) ← (SP) – 2
  ((SP) + 1:(SP)) ← (CS)
  (CS) ← SEG
  (SP) ← (SP) – 2
  ((SP) + 1:(SP)) ← (IP)
(IP) ← DEST
```

Flags Affected None

Description

CALL *procedure-name*

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be *executed* before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The target procedure's relative displacement (up to ±32K) from the CALL instruction is then added to the instruction pointer. This CALL instruction form is "self-relative" and appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are moved together in the same segment.

An intrasegment indirect CALL may be made through memory or a register. SP is decremented by two; IP is pushed onto the stack. The target procedure offset is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and replaced by the offset word in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and replaced by the content of the first word of the doubleword pointer referenced by the instruction.

CALL — CALL PROCEDURE

Encoding

Intra-Segment Direct

1 1 1 0 1 0 0 0	disp-low	disp-high

DEST = (EA)

Intra-Segment Indirect

1 1 1 1 1 1 1 1	mod 0 1 0 r/m

DEST = (IP) + disp

Inter-Segment Direct

1 0 0 1 1 0 1 0	offset-low	offset-high
	seg-low	seg-high

DEST = offset, SEG = seg

Inter-Segment Indirect

1 1 1 1 1 1 1 1	mod 0 1 1 r/m

DEST = (EA), SEG = (EA +2)

CALL Operands	Clocks	Transfers	Bytes	CALL Coding Example
near-proc	19	1	3	CALL NEAR PROC
far-proc	28	2	5	CALL FAR PROC
memptr16	21 + EA	2	2-4	CALL PROC TABLE [SI]
regptr16	16	1	2	CALL AX
memptr32	37 + EA	4	2-4	CALL [BX].TASK [SI]

CBW — CONVERT BYTE TO WORD

Operation if (AL) <80H then (AH) ← 0 else (AH) ← FFH

Flags Affected None

Description CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout register AH. CBW does not affect any flags. CBW can be used to produce a double-length (word) dividend from a byte prior to performing byte division.

Encoding

1 0 0 1 1 0 0 0

CBW Operands	Clocks	Transfers	Bytes	CBW Coding Example
(no operands)	2	–	1	CBW

CLC — CLEAR CARRY

Operation (CF) ← 0

Flags Affected CF

Description CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no c
It (and CMC and STC) is useful in conjunction with the RCL
instructions.

Encoding

1 1 1 1 1 0 0 0

CLC Operands	Clocks	Transfers	Bytes	CLC Coding E:
(no operands)	2	–	1	CLC

CLD — CLEAR DIRECTION FLAG

Operation (DF) ← 0

Flags Affected DF

Description CLD (Clear Direction flag) zeroes DF causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

Encoding

1 1 1 1 1 1 0 0

CLD Operands	Clocks	Transfers	Bytes	CLD Coding Example
(no operands)	2	–	1	CLD

CLI — CLEAR INTERRUPT-ENABLE FLAG

Operation (IF) ← 0

Flags Affected IF

Description CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable is cleared, the 8086 and 8088 do not recognize an external interrupt rec that appears on the INTR line; in other words maskable interrupts are abled. A non-maskable interrupt appearing on NMI line, however, is ored, as is a software interrupt. CLI does not affect any other flags.

Encoding

1 1 1 1 1 0 1 0

CLI Operands	Clocks	Transfers	Bytes	CLI Coding Exampl
(no operands)	2	–	1	CLI

CMC – COMPLEMENT CARRY FLAG

Operation if (CF) = 0 then (CF) ← 1 else (CF) ← 0

Flags Affected CF

Description CMC (Complement Carry flag) "toggles" CF to its opposite state and affects no other flags.

Encoding

1 1 1 1 0 1 0 1

CMC Operands	Clocks	Transfers	Bytes	CMC Coding Example
(no operands)	2	–	1	CMC

CMP – COMPARE

Operation (LSRC) – (RSRC)

Flags Affected AF, CF, OF, PF, SF, ZF

Description **CMP** *destination,source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF, SF and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (jump if greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

Encoding

Memory or Register Operand with Register Operand

0 0 1 1 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA
else LSRC = EA, RSRC = REG

Immediate Operand with Memory or Register Operand

1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data

Immediate Operand with Accumulator

0 0 1 1 1 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data
else LSRC = AX, RSRC = data

CMP Operands	Clocks	Transfers	Bytes	CMP Coding Example
register, register	3	–	2	CMP BX, CX
register, memory	9 + EA	1	2-4	CMP DH, ALPHA
memory, register	9 + EA	1	2-4	CMP [BP+2],SI
register, immediate	4	–	3-4	CMP BL, 02H
memory, immediate	10 + EA	1	3-6	CMP [BX].RADAR [DI],3420H
accumulator, immediate	4	–	2-3	CMP AL, 00010000B

CMPS — COMPARE STRING (BYTE OR WORD)

Operation

```
(LSRC) – (RSRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) – DELTA
  (DI) ← (DI) – DELTA
```

Flags Affected AF, CF, OF, PF, SF, ZF

Description **CMPS** *destination-string,source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates, AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the destination element is greater than the source element. If CMPS is prefixed with REPE or REPZ, the operation is interrupted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)." If CMPS is preceded by REPNE or REPNZ, the operation is interrupted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0)." Thus, CMPS can be used to find matching or differing string elements.

Encoding

1 0 1 0 0 1 1 w

w = 0 then LSRC = (SI), RSRC = (DI), DELTA = 1
else LSRC = (SI) + 1:(SI), RSRC = (DI) + 1:(DI), DELTA = 2

CMPS Operands	Clocks	Transfers	Bytes	CMPS Coding Example
dest-string, source-string	22	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, string-source	9 + 22	2/rep	1	REP COMPS ID, KEY

CWD — CONVERT WORD TO DOUBLEWORD

Operation

```
if (AX) < 8000H then (DX) ← 0
else (DX) ← FFFFH
```

Flags Affected None

Description CWD (Convert Word to Doubleword) extends the sign of the word in register AX throughout register DX. CWD does not affect any flags. CWD can be used to produce a double-length (doubleword) dividend from a word prior to performing word division.

Encoding

1 0 0 1 1 0 0 1

CWD Operands	Clocks	Transfers	Bytes	CWD Coding Example
(no operands)	5	–	1	CWD

DAA — DECIMAL ADJUST FOR ADDITION

Operation

```
if ((AL) & 0FH) >9 or (AF) = 1 then
  (AL) ← (AL) + 6
  (AF) ← 1
if (AL) > 9FH or (CF) = 1 then
  (AL) ← (AL) + 60H
  (CF) ← 1
```

Flags Affected

AF, CF, PF, SF, ZF
OF undefined

Description

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAA.

Encoding

0 0 1 0 0 1 1 1

DAA Operands	Clocks	Transfers	Bytes	DAA Coding Example
(no operands)	4	–	1	DAA

DAS — DECIMAL ADJUST FOR SUBTRACTION

Operation

```
if ((AL) & 0FH) >9 or (AF) = 1 then
  (AL) ← (AL) − 6
  (AF) ← 1
if (AL)>9FH or (CF) = 1 then
  (AL) ← (AL) −60H
  (CF) ← 1
```

Flags Affected

AF, CF, PF, SF, ZF
OF undefined

Description

DAS (Decimal Adjust for Subtraction) corrects the result of a pr
traction of two valid packed decimal operands (the destination op
have been specified as register AL). DAS changes the content
pair of valid packed decimal digits. DAS updates AF, CF, PF, S
the content of OF is undefined following execution of DAS.

Encoding

0 0 1 0 1 1 1 1

DAS Operands	Clocks	Transfers	Bytes	DAS Coding Exam
(no operands)	4	–	1	DAS

DEC — DECREMENT

Operation (DEST) ← (DEST) − 1

Flags Affected AF, OF, PF, SF, ZF

Description DEC (Decrement) subtracts one from the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). DEC updates AF, OF, PF, SF and ZF; it does not affect CF.

Encoding

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

DEST = EA

16-Bit Register Operand

0 1 0 0 1 reg

DEST = REG

DEC Operands	Clocks	Transfers	Bytes	DEC Coding Example
reg16	3	–	1	DEC AX
reg8	3	–	2	DEC AL
memory	15 + EA	2	2-4	DEC ARRAY [SI]

DIV — DIVIDE

Operation

```
(temp) ← (NUMR)
if (temp)/(DIVR) > MAX then the
   following, in sequence
   (QUO),(REM) undefined
   (SP) ← (SP) − 2
   ((SP) + 1:(SP)) ← FLAGS
   (IF) ← 0
   (TF) ← 0
   (SP) ← (SP) − 2
   ((SP) + 1:(SP)) ← (CS)
   (CS) ← (2) i.e., the contents of
      memory locations 2 and 3
   (SP) ← (SP) − 2
   ((SP) + 1:(SP)) ← (IP)
   (IP) ← (0) i.e., the contents of
      locations 0 and 1
else
   (QUO) ← (temp)/(DIVR),where
      / is unsigned division
   (REM) ← (temp) %
(DIVR) where

% is unsigned modulo
```

Flags Affected AF, CF, OF, PF, SF, ZF undefined

Description **DIV** *source*

DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the two-byte dividend assumed to be in registers AL and AH. The byte quotient is returned in AL, and the byte remainder is returned in AH. If the source operand is a word, it is divided into the two-word dividend in registers AX and DX. The word quotient is returned in AX, and the word remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. The content of AF, CF, OF, PF, SF and ZF is undefined following execution of DIV.

Encoding

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM = AH, MAX = FFH
else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = FFFFH

DIV Operands	Clocks	Transfers	Bytes	DIV Coding Example
reg8	80-90	–	2	DIV CL
reg16	144-162	–	2	DIV BX
mem8	86-96 + EA	1	2-4	DIV ALPHA
mem16	154-172 + EA	1	2-4	DIV TABLE [SI]

ESC – ESCAPE

Operation if mod ≠ 11 then data bus ← (EA)

Flags Affected None

Description The ESC (Escape) instruction provides a mechanism by which oth sors (coprocessors) may receive their instructions from the 8086 c struction stream and make use of the 8086 or 8088 addressing m CPU (8086 or 8088) does a no operation (NOP) for the ESC i other than to access a memory operand and place it on the bus.

Encoding

1 1 0 1 1 x	mod x r/m

ESC Operands	Clocks	Transfers	Bytes	ESC Coding Exa
immediate, memory	8+EA	1	2-4	ESC 6, ARRAY
immediate, register	2	–	2	ESC 20, AL

HLT — HALT

Operation None

Flags Affected None

Description HLT (Halt) causes the CPU to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a non-maskable interrupt request on NMI, or, if interrupts are enabled, upon receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

Encoding

1 1 1 1 0 1 0 0

HLT Operands	Clocks	Transfers	Bytes	HLT Coding Example
(no operands)	2	–	1	HLT

IDIV — INTEGER DIVIDE

Operation

```
(temp) ← (NUMR)
if (temp)/(DIVR) > 0 and (temp)
    /(DIVR) > MAX
or (temp)/(DIVR) < 0 and (temp)
    /(DIVR) < 0 – MAX – 1 then
    (QUO), (REM) undefined
    (SP) ← (SP) – 2
    ((SP) + 1:(SP)) ← FLAGS
    (IF) ← 0
    (TF) ← 0
    (SP) ← (SP) – 2
    ((SP) + 1:(SP)) ← (CS)
    (CS) ← (2)
    (SP) ← (SP) – 2
    ((SP) + 1:(SP)) ← (IP)
    (IP) ← (0)
else
    (QUO) ← (temp)/(DIVR), where
      / is signed division
    (REM) ← (temp) % (DIVR) where
      % is signed modulo
```

Flags Affected AF, CF, OF, PF, SF, ZF undefined

Description **IDIV** *source*

IDIV (Integer Divide) performs a signed division of the accumulator (anc extension) by the source operand. If the source operand is a byte, it is vided into the double-length dividend assumed to be in registers AL AH; the single-length quotient is returned in AL, and the single-length mainder is returned in AH. For byte integer division, the maximum posi quotient is +127 (7FH) and the minimum negative quotient is −127 (81 If the source operand is a word, it is divided into the double-length divid in registers AX and DX; the single-length quotient is returned in AX, the single-length remainder is returned in DX. For word integer division, maximum positive quotient is +32,767 (7FFFH) and the minimum nega quotient is −32,767 (8001H). If the quotient is positive and exceeds maximum, or is negative and is less than the minimum, the quotient remainder are undefined, and a type 0 interrupt is generated. In particu this occurs if division by 0 is attempted. Nonintegral quotients are trunca (toward 0) to integers, and the remainder has the same sign as the divide The content of AF, CF, OF, PF, SF and ZF is undefined following IDIV

Encoding

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM = AH, MZX = 7FH
else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = 7FFFH

IDIV Operands	Clocks	Transfers	Bytes	IDIV Coding Example
reg8	101-112	–	2	IDIV BL
reg16	165-184	–	2	IDIV CX
mem8	107-118+EA	1	2-4	IDIV DIVISOR_BYTE[SI]
mem16	171-190+EA	1	2-4	IDIV [BX].DIVISOR_WORD

IMUL — INTEGER MULTIPLY

Operation

(DEST) ← (LSRC) * (RSRC) where
  * is signed multiply
if (ext) = sign-extension of (LOW)
  then (CF) ← 0
else (CF) ← 1;
(OF) ← (CF)

Flags Affected

CF, OF
AF, PF, SF, ZF undefined

Description

IMUL *source*

IMUL (Integer Multiply) performs a signed multiplication of the
erand and the accumulator. If the source is a byte, then it is m
register AL, and the double-length result is returned in AH and
source is a word, then it is multiplied by register AX, and the do
result is returned in registers DX and AX. If the upper half o
(AH for byte source, DX for word source) is not the sign exten
lower half of the result, CF and OF are set; otherwise they a
When CF and OF are set, they indicate that AH or DX contain
digits of the result. The content of AF, PF, SF and ZF is undefine
execution of IMUL.

Encoding

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

if w = 0 then LSRC = AL, RSRC = EA, DEST = AH, EXT = AH, LOW = AL
else LSRC = AX, RSRC = EA, DEST = DX:AX, EXT = DX, LOW = AX

IMUL Operands	Clocks	Transfers	Bytes	IMUL Coding E
reg8	80-98	–	2	IMUL CL
reg16	128-154	–	2	IMUL BX
mem8	86-104+EA	1	2-4	IMUL RATE_BYTE
mem16	134-160+EA	1	2-4	IMUL RATE_WORI

IN — INPUT BYTE OR WORD

Operation (DEST) ← (SRC)

Flags Affected None

Description **IN** *accumulator,port*

IN transfers a byte or a word from an input port to the AL register or the AX register, respectively. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

Encoding

Fixed Port

1 1 1 0 0 1 0 w	port

if w = 0 then SRC = port, DEST = AL
else SRC = port + 1:port, DEST = AX

Variable Port

1 1 1 0 1 1 0 w

if w = 0 then SRC = (DX), DEST = AL
else SRC = (DX)+1:(DX), DEST = AX

IN Operands	Clocks	Transfers	Bytes	IN Coding Example
accumulator, immed8	10	1	2	IN AL, OEAH
accumulator,DX	8	1	1	IN AX, DX

INC — INCREMENT

Operation (DEST) ← (DEST) +1

Flags Affected AF, OF, PF, SF, ZF

Description **INC** *destination*

INC (Increment) adds one to the destination operand. The operand may be byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF and ZF; it does not affect CF.

Encoding

Memory or Register Operand

1 1 1 1 1 1 1 w	mod 0 0 0 r/m

DEST = EA

16-Bit Register Operand

0 1 0 0 0 reg

DEST = REG

INC Operands	Clocks	Transfers	Bytes	INC Coding Example
reg16	3	–	1	INC CX
reg8	3	–	2	INC BL
memory	15+EA	2	2-4	INC ALPHA [DI][BX]

INT — INTERRUPT

Operation

```
(SP) ← (SP) −2
((SP) + 1:(SP))←FLAGS
(IF)←0
(TF)←0
(SP)←(SP) − 2
((SP) + 1:(SP))←(CS)
(CS)←(TYPE * 4 + 2)
(SP)←(SP) − 2
((SP) + 1:(SP))←(IP)
(IP)←(TYPE * 4)
```

Flags Affected IF, TF

Description **INT** *interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap (TF) and interrupt-enable (IF) flags to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word of the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type = 3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as "supervisor calls," i.e., requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

Encoding

1 1 0 0 1 1 0 v	type if v=1

if v = 0 then TYPE = 3
else TYPE = type

INT Operands	Clocks	Transfers	Bytes	INT Coding Example
immed8(type =3)	52	5	1	INT 3
immed8(type ≠ 3)	51	5	2	INT 67

INTO — INTERRUPT ON OVERFLOW

Operation

```
if (OF) = 1 then
  (SP)←(SP) − 2
  ((SP) + 1:(SP))←FLAGS
  (IF)←0
  (TF)←0
  (SP)←(SP) − 2
  ((SP) + 1:(SP))←(CS)
  (CS)←(12H)
  (SP)←(SP) − 2
  ((SP) + 1:(SP))←(IP)
  (IP)←(10H)
```

Flags Affected None

Description INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt procedure (its type is 4) through the interrupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

Encoding

1 1 0 0 1 1 1 0

INTO Operands	Clocks	Transfers	Bytes	INTO Coding Example
(no operands)	53 or 4	5	1	INTO

IRET — INTERRUPT RETURN

Operation

(IP)←((SP)+1:(SP))
(SP)←(SP)+2
(CS)←((SP)+1:(SP))
(SP)←(SP)+2
FLAGS←((SP)+1:(SP))
(SP)←(SP)+2

Flags Affected All

Description IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

Encoding

1 1 0 0 1 1 1 1

IRET Operands	Clocks	Transfers	Bytes	IRET Coding Example
(no operands)	32	3	1	IRET

JA — JUMP ON ABOVE
JNBE — JUMP ON NOT BELOW OR EQUAL

Operation

if (CF) & (ZF) = 0 then
(IP) ← (IP) + disp (sign-extended to 16-bits)

Flags Affected None

Description Jump on Above (JA)/Jump on Not Below or Equal (JNBE) trans
to the target operand (IP + displacement) if CF and ZF = 0.

Encoding

0 1 1 1 0 1 1 1	disp

JA/JNBE Operands	Clocks	Transfers	Bytes	JA Coding Ex
short-label	16 or 4	–	2	JA ABOVI
				JNBE Coding E
				JNBE ABO

JAE – JUMP ON ABOVE OR EQUAL
JNB – JUMP ON NOT BELOW

Operation if (CF) = 0 then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected None

Description JAE (Jump on Above or Equal)/JNB (Jump on Not Below) transfers control to the target operand (IP + displacement) if CF = 0.

Encoding

0 1 1 1 0 0 1 1	disp

JAE/JNB Operands	Clocks	Transfers	Bytes	JAE Coding Example
short-label	16 or 4	–	2	JAE ABOVE_EQUAL

JB – JUMP ON BELOW
JNAE – JUMP ON NOT ABOVE OR EQUAL

Operation if (CF) = 1 then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected None

Description JB (Jump on Below)/JNAE (Jump on Not Above or Equal) transfers control to the target operand (IP + displacement) if CF = 1.

Encoding

0 1 1 1 0 0 1 0	disp

JB/JNAE Operands	Clocks	Transfers	Bytes	JB Coding Example
short-label	16 or 4	–	2	JB BELOW

JBE — JUMP ON BELOW OR EQUAL
JNA — JUMP ON NOT ABOVE

Operation

if (CF) or (ZF) = 1 then
(IP) ← (IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

JBE (Jump on Below or Equal)/JNA (Jump on Not Above) transfers control to the target operand (IP + displacement) if CF or ZF = 1.

Encoding

0 1 1 1 0 1 1 0	disp

JBE/JNA Operands	Clocks	Transfers	Bytes	JNA Coding Example
short-label	16 or 4	–	2	JNA NOT_ABOVE

JC — JUMP ON CARRY

Operation

if (CF) = 1 then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

JC (Jump on Carry) transfers control to the target operand (IP - ment) on the condition CF=1.

Encoding

0 1 1 1 0 0 1 0	disp

JC Operands	Clocks	Transfers	Bytes	JC Coding Ex
short-label	16 or 4	–	2	JC CARRY_SE

JCXZ – JUMP IF CX REGISTER ZERO

Operation if (CX) = 0 then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected None

Description **JCXZ** *short-label*

JCXZ (Jump if CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

Encoding

1 1 1 0 0 0 1 1	disp

JCXZ Operands	Clocks	Transfers	Bytes	JCXZ Coding Example
short-label	18 or 6	–	2	JCXZ COUNT DONE

JE – JUMP ON EQUAL
JZ – JUMP ON ZERO

Operation if (ZF) = 1 then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected None

Description JE (Jump on Equal)/JZ (Jump on Zero) transfers control to the target operand (IP + displacement) if ZF = 1.

Encoding

0 1 1 1 0 1 0 0	disp

JE/JZ Operands	Clocks	Transfers	Bytes	JZ Coding Example
short-label	16 or 4	–	2	JZ ZERO

JG – JUMP ON GREATER
JNLE – JUMP ON NOT LESS OR EQUAL

Operation

if ((SF) = (OF)) or (ZF) = 0 then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

JG (Jump on Greater Than)/JNLE (Jump on Not Less Than or Equal) transfers control to the target operand (IP + displacement) if the conditions ((SF XOR OF) or ZF = 0) are greater than/not less than or equal to the tested value.

Encoding

0 1 1 1 1 1 1 1	disp

JG/JNLE Operands	Clocks	Transfers	Bytes	JG Coding Example
short-label	16 or 4	–	2	JG GREATER

JGE – JUMP ON GREATER OR EQUAL
JNL – JUMP ON NOT LESS

Operation

if (SF) = (OF) then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

JGE (Jump on Greater Than or Equal)/JNL (Jump on Not Less Than) transfers control to the target operand (IP + displacement) if the condition (SF XOR OF = 0) is greater than or equal/not less than the tested value.

Encoding

0 1 1 1 1 1 0 1	disp

JGE/JNL Operands	Clocks	Transfers	Bytes	JGE Coding Example
short-label	16 or 4	–	2	JGE GREATER EQUAL

JL – JUMP ON LESS
JNGE – JUMP ON NOT GREATER OR EQUAL

Operation

if (SF) ≠ (OF) then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

JL (Jump on Less Than)/JNGE (Jump on Not Greater Than or Equal), transfers control to the target operand if the condition (SF XOR OF = 1) is less than/not greater than or equal to the tested value.

Encoding

0 1 1 1 1 1 0 0	disp

JL/JNGE Operands	Clocks	Transfers	Bytes	JL Coding Example
short-label	16 or 4	–	2	JL LESS

JLE – JUMP ON LESS OR EQUAL
JNG – JUMP ON NOT GREATER

Operation

if ((SF) ≠ (OF)) or ((ZF) = 1) then
(IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

JLE (Jump on Less Than or Equal to)/JNG (Jump on Not Greater Than) transfers control to the target operand (IP + displacement) if the conditions tested ((SF XOR OF) or ZF = 1) are less than or equal to/not greater than the tested value.

Encoding

0 1 1 1 1 1 1 0	disp

JLE/JNG Operands	Clocks	Transfers	Bytes	JNG Coding Example
short-label	16 or 4	–	2	JNG NOT GREATER

JMP — JUMP UNCONDITIONALLY

Operation if Inter-Segment then (CS)←SEG
(IP)←DEST

Flags Affected None

Description **JMP** *target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack; no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP), or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte instruction form called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within ±32K. Intrasegment direct JMPS are self-relative and appropriate in position-independent (dynamically relocatable) routines in which the JMP and its target are moved together in the same segment.

An intrasegment indirect JMP may be made either through memory or a 16-bit general register. In the first case, the word content referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP and the second word replaces CS.

Encoding

Intra-Segment Direct

1 1 1 0 1 0 0 1	disp-low	disp-high

DEST = (IP) + disp

Intra-Segment Direct Short

1 1 1 0 1 0 1 1	disp

DEST = (IP) + disp sign extended to 16-bits

Intra-Segment Indirect

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

DEST = (EA)

Inter-Segment Direct

1 1 1 0 1 0 1 0	offset-low	offset-high
	seg-low	seg-high

DEST = offset, SEG = seg

Inter-Segment Indirect

1 1 1 1 1 1 1 1	mod 1 0 1 r/m

DEST = (EA), SEG = (EA + 2)

JMP Operands	Clocks	Transfers	Bytes	JMP Coding Example
short-label	15	–	2	JMP SHORT
near-label	15	–	3	JMP WITHIN SEGMENT
far-label	15	–	5	JMP FAR LABEL
memptr16	18 + EA	–	2-4	JMP[BX].TARGET
regptr16	11	–	2	JMP CX
memptr32	24 + EA	–	2-4	JMP OTHER.SEG[SI]