JNC — JUMP ON NOT CARRY

Operation

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

Flags Affected

None

Description

JNC (Jump on Not Carry) transfers control to the target operand (IP + displacement) on the condition CF = 0.

Encoding

0 1 1 1 0 0 1 1	disp

JNC Operands	Clocks	Transfers	Bytes	JNC Coding Example
short-label	16 or 4	–	2	JNC NO_CARRY

JNE — JUMP ON NOT EQUAL
JNZ — JUMP ON NOT ZERO

Operation

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

Flags Affected

None

Description

JNE (Jump on Not Equal to)/JNZ (Jump on Not Zero) transfers control to the target operand (IP + displacement) if the condition tested (ZF = 0) is true.

Encoding

0 1 1 1 0 1 0 1	disp

JNE/JNZ Operands	Clocks	Transfers	Bytes	JNE Coding Example
short-label	16 or 4	–	2	JNE NOT_EQUAL

JNO — JUMP ON NOT OVERFLOW

Operation

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

Flags Affected

None

Description

JNO (Jump on Not Overflow) transfers control to the target operand (IP + displacement) if the condition tested (OF = 0) is true.

Encoding

0 1 1 1 0 0 0 1	disp

JNO Operands	Clocks	Transfers	Bytes	JNO Coding Example
short-label	16 or 4	–	2	JNO NO_OVERFLOW

JNS — JUMP ON NOT SIGN

Operation

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

Flags Affected

None

Description

JNS (Jump on Not Sign) transfers control to the target operand placement) when the tested condition (SF = 0) is true.

Encoding

0 1 1 1 1 0 0 1	disp

JNS Operands	Clocks	Transfers	Bytes	JNS Coding E
short-label	16 or 4	–	2	JNS POSITIVE

JP – JUMP ON PARITY
JPE – JUMP ON PARITY EQUAL

Operation

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

Flags Affected

None

Description

JP (Jump on Parity)/JPE (Jump on Parity Equal) transfers control to the target operand (IP + displacement) if the condition tested (PF = 1) is true.

Encoding

0 1 1 1 1 0 1 0	disp

JP/JPE Operands	Clocks	Transfers	Bytes	JPE Coding Example
short-label	16 or 4	–	2	JPE EVEN_PARITY

JS – JUMP ON SIGN

Operation

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

Flags Affected

None

Description

JS (Jump on Sign) transfers control to the target operand (IP + displacement) if the tested condition (SF = 1) is true.

Encoding

0 1 1 1 1 0 0 0	disp

JS Operands	Clocks	Transfers	Bytes	JS Coding Example
short-label	16 or 4	–	2	JS NEGATIVE

LAHF – LOAD REGISTER AH FROM FLAGS

Operation (AH)←(SF):(ZF):X:(AF):X:(PF):X:(CF)

Flags Affected None

Description LAHF (load register AH from flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH. The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

Encoding

1 0 0 1 1 1 1 1

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

LDS – LOAD POINTER USING DS

Operation (REG)←(EA)
(DS)←(EA +2)

Flags Affected None

Description **LDS** *destination,source*

LDS (load pointer using DS) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

Encoding

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

if mod = 11 then undefined operation

LDS Operands	Clocks	Transfers	Bytes	LDS Coding Example
reg16,mem32	16+EA	2	2-4	LDS SI, DATA.SEG[DI]

LEA — LOAD EFFECTIVE ADDRESS

Operation (REG)←EA

Flags Affected None

Description **LEA** *destination,source*

LEA (load effective address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general register. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to load these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

Encoding

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

if mod = 11 then undefined operation

LEA Operands	Clocks	Transfers	Bytes	LEA Coding Example
reg16,mem16	2+EA	—	2-4	LEA BX,[BP][DI]

LES — LOAD POINTER USING ES

Operation (REG)←(EA)
(ES)←(EA + 2)

Flags Affected None

Description **LES** *destination,source*

LES (load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

Encoding

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

if mod = 11 then undefined operation

LES Operands	Clocks	Transfers	Bytes	LES Coding Example
reg16,mem32	16+EA	2	2-4	LES DI,[BX].TEXT_BUFF

LOCK — LOCK THE BUS

Operation None

Flags Affected None

Description LOCK is a one-byte prefix that causes the 8088 (configured in maximum mode) to assert its bus LOCK signal while the following instruction executes. LOCK does not affect any flags.

The instruction most useful in this context is an exchange register with memory. A simple software lock may be implemented with the following code sequence:

```
Check   MOV    AL,1      ;set AL to 1 (implies locked)
 LOCK   XCHG   Sema,AL   ;test and set lock
        TEST   AL,AL     ;set flags based on AL
        JNZ    Check     ;retry if lock already set
        •
        •
        MOV    Sema,0    ;clear the lock when done
```

The LOCK prefix may be combined with the segment override and/or REP prefixes.

Encoding

1 1 1 1 0 0 0 0

LOCK Operands	Clocks	Transfers	Bytes	LOCK Coding Example
(no operands)	2	–	1	LOCK XCHG FLAG, AL

LODS — LOAD STRING (BYTE OR WORD)

Operation

(DEST)←(SRC)
if (DF) = 0 then (SI)←(SI) + DELTA
else (SI)←(SI) − DELTA

Flags Affected None

Description **LODS** *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

Encoding

1 0 1 0 1 1 0 w

if w = 0 then SRC = (SI), DEST = AL, DELTA = 1
else SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

LODS Operands	Clocks	Transfers	Bytes	LODS Coding Example
source-string	12	1	1	LODS CUSTOMER_NAME
(repeat) source-string	9 + 13	1/rep	1	REP LODS NAME

LOOP – LOOP

Operation

(CX)←(CX) – 1
if (CX)≠0 then
  (IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

LOOP *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

Encoding

1 1 1 0 0 0 1 0	disp

LOOP Operands	Clocks	Transfers	Bytes	LOOP Coding Example
short-label	17 or 5	–	2	LOOP AGAIN

LOOPE – LOOP WHILE EQUAL
LOOPZ – LOOP WHILE ZERO

Operation

(CX)←(CX) – 1
if (ZF) = 1 and (CX) ≠ 0 then
  (IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

LOOPE/LOOPZ *short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE/LOOPZ is executed.

Encoding

1 1 1 0 0 0 0 1	disp

LOOPE/LOOPZ Operands	Clocks	Transfers	Bytes	LOOPE Coding Example
short-label	18 or 6	–	2	LOOPE AGAIN

LOOPNZ – LOOP WHILE NOT ZERO
LOOPNE – LOOP WHILE NOT EQUAL

Operation

(CX)←(CX) – 1
if (ZF) = 0 and (CX) ≠ 0 then
  (IP)←(IP) + disp (sign-extended to 16-bits)

Flags Affected

None

Description

LOOPNE/LOOPNZ *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed.

Encoding

1 1 1 0 0 0 0 0	disp

LOOPNE/LOOPNZ Operands	Clocks	Transfers	Bytes	LOOPNE Coding Example
short-label	19 or 5	–	2	LOOPNE AGAIN

MOV – MOVE (BYTE OR WORD)

Operation

(DEST)←(SRC)

Flags Affected

None

Description

MOV *destination,source*

MOVE transfers a byte or a word from the source operand to the destination operand.

Encoding

Memory or Register Operand to/from Register Operand

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

if d = 1 then SRC = EA, DEST = REG
else SRC = REG, DEST = EA

Immediate Operand to Memory or Register Operand

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

SRC = data, DEST = EA

Immediate Operand to Register

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

SRC = data, DEST = REG

Memory Operand to Accumulator

1 0 1 0 0 0 0 w	addr-low	addr-high

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

Accumulator to Memory Operand

1 0 1 0 0 0 1 w	addr-low	addr-high

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

Memory or Register Operand to Segment Register

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

if reg ≠ 01 then SRC = EA, DEST = REG
else undefined operation

Segment Register to Memory or Register Operand

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

SRC = REG,DEST = EA

MOV Operands	Clocks	Transfers	Bytes	MOV Coding Example
memory,accumulator	10	1	3	MOV ARRAY AL
accumulator,memory	10	1	3	MOV AX, TEMP_RESULT
register,register	2	–	2	MOV AX, 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,reg16	2	–	2	MOV ES, CX
seg-reg,mem16	8+EA	1	2-4	MOV DS, SEGMENT_BASE
reg16,seg-reg	2	–	2	MOV BP, SS
memory,seg-reg	9+EA	1	2-4	MOV[BX]SEG_SAVE,CX

MOVS – MOVE STRING

Operation (DEST)←(SRC)

Flags Affected None

Description **MOVS** *destination-string,source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

Encoding

1 0 1 0 0 1 0 w

if w = 0 then SRC = (SI), DEST = AL, DELTA = 1
else SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

MOVS Operands	Clocks	Transfers	Bytes	MOVS 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

MUL — MULTIPLY

Operation

(DES)←(LSRC) * (RSRC), where * is unsigned multiply
if (EXT) = 0 then (CF)←0
else (CF)←1;
(OF)←(CF)

Flags Affected

CF, OF
AF, PF, SF, ZF undefined

Description

MUL *source*

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is non-zero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of MUL.

Encoding

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

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

MUL Operands	Clocks	Transfers	Bytes	MUL Coding Example
reg8	70-77	–	2	MUL BL
reg16	118-113	–	2	MUL CX
mem8	76-83 + EA	1	2-4	MUL MONTH[SI]
mem16	124-139 + EA	1	2-4	MUL BAUD RATE

NEG — NEGATE

Operation

(EA)←SRC – (EA)
(EA)←(EA) + 1 (affecting flags)

Flags Affected

AF, CF, OF, PF, SF, ZF

Description

NEG *destination*

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an integer. If the operand is zero, its sign is not changed. Attempting to negate a byte containing –128 or a word containing –32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF and ZF. CF is always set except when the operand is zero, in which case it is cleared.

Encoding

1 1 1 1 0 1 1 w	mov 0 1 1 r/m

if w = 0 then SRC = FFH
else SRC = FFFFH

NEG Operands	Clocks	Transfers	Bytes	NEG Coding Example
register	3	–	2	NEG AL
memory	16+EA	2	2-4	NEG MULTIPLIER

NOP — NO OPERATION

Operation None

Flags Affected None

Description **NOP**

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

Encoding

1 0 0 1 0 0 0 0

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

NOT — LOGICAL NOT

Operation (EA)←SRC – (EA)

Flags Affected None

Description **NOT** *destination*

NOT inverts the bits (forms the one's complement) of the by operand.

Encoding

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

if w = 0 then SRC = FFH
else SRC = FFFFH

NOT Operands	Clocks	Transfers	Bytes	NOT Coding E
register	3	—	—	NOT AX
memory	16 + EA	2	—	NOT CHARAC

POP — POP

Operation (DEST)←((SP)+1:(SP))
(SP)←(SP)+2

Flags Affected None

Description **POP** *destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

Encoding

Memory or Register Operand

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

DEST = EA

Register Operand

0 1 0 1 1 reg

DEST = REG

Segment Register

0 0 0 reg 1 1 1

if reg ≠ 01 then DEST = REG
else undefined operation

POP Operands	Clocks	Transfers	Bytes	POP 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 — POP FLAGS

Operation Flags←((SP)+1:(SP))
(SP)←(SP)+2

Flags Affected All

Description **POPF**

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (see Figure 2-32). SP is then incremented by two to point to the new top of stack. PUSHF and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory-image and then popping the flags.

Encoding

1 0 0 1 1 1 0 1

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

PUSH – PUSH

Operation (SP)←(SP) – 2
((SP) + 1:(SP))←(SRC)

Flags Affected None

Description **PUSH** *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

Encoding

Memory or Register Operand

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

SRC = EA

Register Operand

0 1 0 1 0 reg

SRC = REG

Segment Register

0 0 0 reg 1 1 0

SRC = REG

PUSH Operands	Clocks	Transfers	Bytes	PUSH 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 – PUSH FLAGS

Operation (SP)←(SP) – 2
((SP) + 1:(SP))← Flags

Flags Affected None

Description **PUSHF**

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to by SP. The flags themselves are not affected.

Encoding

1 0 0 1 1 1 0 0

PUSHF Operands	Clocks	Transfers	Bytes	PUSHF Coding Example
(no operands)	10	1	1	PUSHF

RCL — ROTATE THROUGH CARRY LEFT

Operation

```
(temp)←COUNT
do while (temp) ≠ 0
  (tmpcf)←(CF)
  (CF)←high-order bit of (EA)
  (EA)←(EA) * 2 + (tmpcf)
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF)←1
  else (OF)←0
else (OF) undefined
```

Flags Affected CF, OF

Description **RCL** *destination,count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and itself is replaced by the high-order bit of the destination.

Encoding

Rotate by 1 or CL:

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

if v = 0 then COUNT = 1
else COUNT = (CL)

RCL Operands	Clocks	Transfers	Bytes	RCL 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].PARAM, CL

RCR — ROTATE THROUGH CARRY RIGHT

Operation

```
(temp)←COUNT
do while (temp)≠0
  (tmpcf)←(CF)
  (CF)←low-order bit of (EA)
  (EA)←(EA)/2
  high-order bit of (EA)←(tmpcf)
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA)≠next-to-high-order bit of (EA)
  then (OF)←1
  else(OF)←0
else (OF) undefined
```

Flags Affected CF, OF

Description **RCR** *destination,count*

RCR (Rotate through Carry Right) operates exactly like RCL e the bits are rotated right instead of left.

Encoding

Rotate by 1 or CL:

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

if v = 0 then COUNT = 1
else COUNT = (CL)

RCR Operands	Clocks	Transfers	Bytes	RCR Coding E:
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].STAT
memory, CL	20+4/bit+EA	2	2-4	RCR ARRAY[DI]

REP — REPEAT
REPE/REPZ — REPEAT WHILE EQUAL/REPEAT WHILE ZERO
REPNE/REPNZ — REPEAT WHILE NOT EQUAL/REPEAT WHILE NOT ZERO

Operation

```
do while (CX) ≠ 0
  service pending interrupts (if any) execute primitive string operation in
    succeeding byte
  (CX)←(CX) −1
  if primitive operation is CMPB, CMPW, SCAB, or SCAW and (ZF)≠z
    then exit from while loop
```

Flags Affected None

Description **REP/REPE/REPZ/REPNE/REPNZ**

Repeat, Repeat While Equal, Repeat While Zero, Repeat While Not Equal and Repeat While Not Zero are mnemonics for two forms of the prefix byte that controls subsequent string instruction repetition. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ except that the zero flag must be cleared or the repetition is terminated. ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. However, execution does *not* resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. At interrupt time, the processor "remembers" only the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a non-maskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

Encoding

1 1 1 1 0 0 1 z

REP Operands	Clocks	Transfers	Bytes	REP Coding Example
(no operands)	2	–	1	REP MOVS DEST, SRCE
REPE/REPZ Operands	**Clocks**	**Transfers**	**Bytes**	**REPE Coding Example**
(no operands)	2	–	1	REPE CMPS DATA, KEY
REPNE/REPNZ Operands	**Clocks**	**Transfers**	**Bytes**	**REPNE Coding Example**
(no operands)	2	–	1	REPNE SCAS INPUT_LINE

RET — RETURN

Operation

```
(IP)←((SP)=1:(SP))
(SP)←(SP) +2
if Inter-Segment then
  (CS)←((SP)+1:(SP))
  (SP)←(SP)+2
if Add Immediate to Stack Pointer
  then (SP)←(SP) + data
```

Flags Affected None

Description **RET** *optional-pop-value*

RET (Return transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intrasegment RET if the programmer has defined the procedure NEAR, or an intersegment RET if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

Encoding

Intra-Segment

1 1 0 0 0 0 1 1

Intra-Segment and Add Immediate to Stack Pointer

1 1 0 0 0 0 1 0	data-low	data-high

Inter-Segment

1 1 0 0 1 0 1 1

Inter-Segment and Add Immediate to Stack Pointer

1 1 0 0 1 0 1 0	data-low	data-high

RET Operands	Clocks	Transfers	Bytes	RET 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 — ROTATE LEFT

Operation

```
(temp)←COUNT
do while (temp)≠0
  (CF)←high-order bit of (EA)
  (EA)←(EA) * 2 + (CF)
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF)←1
  else (OF)←0
else (OF) undefined
```

Flags Affected CF, OF

Description **ROL** *destination,count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

Encoding

Rotate by 1 or CL:

1 1 0 1 0 0 v w	mov 0 0 0 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

ROL Operands	Clocks	Transfers	Bytes	ROL 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 — ROTATE RIGHT

Operation

```
(temp)←COUNT
do while (temp)≠0
  (CF)←low-order bit of (EA)
  (EA)←(EA)/2
  high-order bit of (EA)←(CF)
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA)≠next-to-high-order bit of (EA)
then (OF)←1
  else (OF)←0
else (OF) undefined
```

Flags Affected CF, OF

Description **ROR** *destination,count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

Encoding

Rotate by 1 or CL:

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

if v = 0 then COUNT = 1
else COUNT = (CL)

ROR Operands	Clocks	Transfers	Bytes	ROR Coding Example
register, 1	2	–	2	ROR AL, 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 — STORE REGISTER AH INTO FLAGS

Operation (SF):(ZF):X:(AF):X:(PF):X:(CF)←(AH)

Flags Affected AF, CF, PF, SF, ZF

Description **SAHF**

SAHF (store register AH into flags) transfers bits 7, 6, 4, 2 and register AH into SF, ZF, AF, PF and CF, respectively, replacing v values these flags previously had. OF, DF, IF and TF are not affect instruction is provided for 8080/8085 compatibility.

Encoding

1 0 0 1 1 1 1 0

SAHF Operands	Clocks	Transfers	Bytes	SAHF Coding Exar
(no operands)	4	–	1	SAHF

SAL — SHIFT ARITHMETIC LEFT
SHL—SHIFT LOGICAL LEFT

Operation

```
(temp)←COUNT
do while (temp)≠0
  (CF)←high-order bit of (EA)
  (EA)←(EA) * 2
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA)≠(CE)
    then (OF)←1
  else (OF)←0
else (OF) undefined
```

Flags Affected

CF, OF, PF, SF, ZF
AF undefined

Description

SHL/SAL *destination,count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then OF is cleared.

Encoding

Shift by 1 or CL:

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

if v = 0 then COUNT = 1
else COUNT = (CL)

SAL/SHL Operands	Clocks	Transfers	Bytes	SAL/SHL Coding Example
register, 1	2	—	2	SAL AH, 1
register, CL	8+4/bit	—	2	SHL DI, CL
memory, 1	15+EA	2	2-4	SHL[BX].OVERDRAW, 1
memory, CL	20+4/bit+EA	2	2-4	SAL STORE_COUNT, CL

SAR — SHIFT ARITHMETIC RIGHT

Operation

```
(temp)←COUNT
do while (temp)≠0
  (CF)←low-order bit of (EA)
  (EA)←(EA)/2, where / is equivalent to signed division, roundi
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA)≠next-to-high-order bit of (EA)
    then (OF)←1
else (OF)←0
else (OF)←0
```

Flags Affected

CF, OF, PF, SF, ZF
AF undefined

Description

SAR *destination,count*

SAR (Shift Arithmetic Right) shifts the bits in the destination ope or word) to the right by the number of bits specified in the coun Bits equal to the original high-order (sign) bit are shifted in o preserving the sign of the original value. Note that SAR does nc the same result as the dividend of an "equivalent" IDIV instruc destination operand is negative and 1-bits are shifted out. For exan ing −5 right by one bit yields −3, while integer division −5 by 2 The difference in the instructions is that IDIV truncates all numb zero, while SAR truncates positive numbers toward zero and neg bers toward negative infinity.

Encoding

Shift by 1 or CL:

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

if v = 0 then COUNT = 1
else COUNT = (CL)

SAR Operands	Clocks	Transfers	Bytes	SAR Coding E
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_BLOCK
memory, CL	20+4/bit+EA	2	2-4	SAR N_BLOCK

SBB — SUBTRACT WITH BORROW

Operation

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

Flags Affected

AF, CF, OF, PF, SF, ZF

Description

SBB *destination,source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF, and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

Encoding

Memory or Register Operand and Register Operand

0 0 0 1 1 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 from Memory or Register Operand

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

LSRC = EA, RSRC = data, DEST = EA

Immediate Operand from Accumulator

0 0 0 1 1 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

SBB Operands	Clocks	Transfers	Bytes	SBB 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, 2
memory, immediate	17+EA	2	3-6	SBB COUNT [SI], 10

SCAS — SCAN (BYTE OR WORD) STRING

Operation

(LSRC) − (RSRC)
if (DF) = 0 then (DI) ← (DI) + DELTA
else (DI) ← (DI) − DELTA

Flags Affected

AF, CF, OF, PF, SF, ZF

Description

SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.

Encoding

1 0 1 0 1 1 1 w

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

SCAS Operands	Clocks	Transfers	Bytes	SCAS Coding Example
dest-string	15	1	1	SCAS INPUT_LINE
dest-string	9+15	1/rep	1	REPNE SCAS BUFFER

SHR — SHIFT LOGICAL RIGHT

Operation

```
(temp)←COUNT
do while (temp)≠0
  CF)←low-order bit of (EA
  (EA)←(EA)/2, where / is equivalent to unsigned division
  (temp)←(temp) −1
if COUNT = 1 then
  if high-order bit of (EA)≠next-to-high-order bit of (EA)
    then (OF)←1
  else (OF)←0
else (OF) undefined
```

Flags Affected

CF, OF, PF, SF, ZF
AF undefined

Description

SHR *destination,source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

Encoding

Shift by 1 or CL:

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

if v = 0 then COUNT = 1
else COUNT = (CL)

SHR Operands	Clocks	Transfers	Bytes	SHR 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

STC — SET CARRY

Operation

(CF)←1

Flags Affected

CF

Description

STC

STC (Set Carry flag) sets CF to 1 and affects no other flags.

Encoding

1 1 1 1 1 0 0 1

STC Operands	Clocks	Transfers	Bytes	STC Coding I
(no operands)	2	–	1	STC

STD — SET DIRECTION FLAG

Operation (DF)←1

Flags Affected DF

Description **STD**

STD (Set Direction flag) sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers. STD does not affect any other flags.

Encoding

1 1 1 1 1 1 0 1

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

STI — SET INTERRUPT ENABLE FLAG

Operation (IF)←1

Flags Affected IF

Description **STC**

STI (Set Interrupt-enable flag) sets IF to 1, enabling processor rec maskable interrupt requests appearing on the INTR line. Note hov a pending interrupt will not actually be recognized until the instr lowing STI has executed. STI does not affect any other flags.

Encoding

1 1 1 1 1 0 1 1

STI Operands	Clocks	Transfers	Bytes	STI Coding Ex
(no operands)	2	–	1	STI

STOS — STORE (BYTE/OR/WORD) STRING

Operation

(DEST)←(SRC)
if(DF) = 0 then (DI)←(DI) + DELTA
else (DI)←(DI) −DELTA

Flags Affected

None

Description

STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation. STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

Encoding

1 0 1 0 1 0 1 w

if w = 0 then SRC = AL, DEST = (DI), DELTA = 1
else SRC = AX, DEST = (DI) +1:(DI), DELTA = 2

STOS Operands	Clocks	Transfers	Bytes	STOS Coding Example
dest-string	11	1	1	STOP PRINT_LINE
(repeat) dest-string	9+10/rep	1/rep	1	REP STOS DISPLAY

SUB — SUBTRACT

Operation

(DEST)←(LSRC) − (RSRC)

Flags Affected

AF, CF, OF, PF, SF, ZF

Description

SUB *destination,source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF and ZF.

Encoding

Memory or Register Operand and Register Operand

0 0 1 0 1 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 from Memory or Register Operand

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

LSRC = EA, RSRC = data, DEST = EA

Immediate Operand from Accumulator

0 0 1 0 1 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

SUB Operands	Clocks	Transfers	Bytes	SUB 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

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

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

Description **TEST** *destination,source*

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result, i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (jump if not zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.

Encoding

Memory or Register Operand with Register Operand

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

LSRC = REG, RSRC = EA

Immediate Operand with Memory or Register Operand

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

LSRC = EA, RSRC = data

Immediate Operand with Accumulator

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

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

TEST Operands	Clocks	Transfers	Bytes	TEST Coding Example
register, register	3	–	2	TEST SI, DI
register, memory	9+EA	1	2-4	TEST SI, END_
accumulator, immediate	4	–	2-3	TEST AL, 00100000B
register, immediate	5	–	3-4	TEST BX, 0CC4H
memory, immediate	11+EA	–	3-6	TEST RETURN_CODE,01H

WAIT – WAIT

Operation None

Flags Affected None

Description WAIT causes the CPU to enter the wait state while its TEST l active. WAIT does not affect any flags.

Encoding

1 0 0 1 1 0 1 1

WAIT Operands	Clocks	Transfers	Bytes	WAIT Coding E
(no operands)	4+5n	–	1	WAIT

XCHG — EXCHANGE

Operation (temp)←(DEST)
(DEST)←(SRC)
(SRC)←(temp)

Flags Affected None

Description **XCHG** *destination,source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors (see the section "Minimum and Maximum Modes" in Chapter 2).

Encoding

Memory or Register Operand with Register Operand

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

SRC = EA, DEST = REG

Register Operand with Accumulator

1 0 0 1 0 reg

SRC = REG, DEST = AX

XCHG Operands	Clocks	Transfers	Bytes	XCHG Coding Example
accumulator, reg16	3	–	1	XCHG AX, BX
memory, register	17+EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	–	2	XCHG AL, BL

XLAT — TRANSLATE

Operation AL←((BX)+(AL))

Flags Affected None

Description **XLAT** *translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

Encoding

1 1 0 1 0 1 1 1

XLAT Operands	Clocks	Transfers	Bytes	XLAT Coding Example
source-table	11	1	1	XLAT ASCII_TAB

XOR — EXCLUSIVE OR

Operation

(DEST)←(LSRC) XOR (RSRC)
(CF)←0
(OF)←0

Flags Affected

CF, OF, PF, SF, ZF
AF undefined

Description

XOR *destination,source*

XOR (Exclusive Or) performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

Encoding

Memory or Register Operand with Register Operand

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

LSRC = EA, RSRC = data, DEST = EA

Immediate Operand to Accumulator

0 0 1 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

XOR Operands	Clocks	Transfers	Bytes	XOR 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, 01000010B
register,immediate	4	–	3-4	XOR SI, 00C2H
memory,immediate	17+EA	2	3-6	XOR RETURN_CODE,0D2H

8086/8088 PROGRAMMING EXAMPLES

In this section and the section following, specific programming examples are provided which illustrate how the instruction set and addressing modes may be used in various commonly encountered programming situations.

The programs are primarily written in ASM-86. ASM-86 is the 8086 assembly language. It provides the programmer who is familiar with the CPU architecture, access to all processor features. For critical code segments within programs that make sophisticated use of the hardware, have extremely demanding performance or memory constraints, ASM-86 is the best choice. For detailed information about Intel's 8086 assembly language see: *ASM86 Language Reference Manual*, 121703.

Programs can also be written in high-level languages such as PL/M-86. PL/M-86 is a high-level language suitable for most microprocessor applications. It is easy to use, even by programmers who have little experience with microprocessors. Because it reduces software development time, PL/M-86 is ideal for most of the programming in any application, especially applications that must get to market quickly.

The languages are completely compatible, and a judicious combination of the two often makes good sense. Prototype software can be developed rapidly with a high-level language. When the system is operating correctly, it can then be analyzed to see which sections can best profit from being written in ASM-86. Since the logic of these sections has already been debugged, selective rewriting can be done quickly and with low risk.

The programming examples in this section address the following topics:

- Procedures
- JMP and CALL (jump, call)
- Bit manipulation
- Dynamic code relocation
- Memory mapped I/O
- Breakpoints
- Interrupt handling
- String operations

The examples are intended to show one way to use the instruction set and addressing modes. They do not demonstrate the "best" way to solve a particular problem. The flexibility of the 8086 application differences plus variations in programming style usually add up to a number of ways to implement a programming solution.

Procedures (parameters, reentrancy)

The code in Figure 3-30 illustrates several techniques that are typically used in writing ASM-86 procedures. In this example a calling program invokes a procedure (called EXAMPLE) twice, passing it a different byte array each time. Two parameters are passed