Arquitectura de Computadores

Conjunto de Instruções da família de μP Intel 80x86

Versão 1.0

(Retirado de Norton Guide)

Departamento de Engenharia Informática

ISEP - IPP

Junho de 1996

OVERVIEW

Overview of the Instruction Format

See Also: EA, Flags

Each entry in this list includes information on which flags in the 8088's flag register are changed, and how they're changed. Since there are 9 flags in the flags register, the flags display is very compact:

Flags:	O	D	I	T	S	Z	A	P	C
			0		*	*	?	*	0

? Undefined after the operation.
* Changed to reflect the results of the instruction.
0 Always cleared
1 Always set

The timing charts show timings for the 8088. Since the 80x8x processors execute instructions in fewer clock cycles than the 8088, these charts represent the worst case.

Operands	This field gives the list of possible operands and addressing modes for each instruction.
Clocks	Number of clock cycles required to execute the instruction on an 8088. Effective Address calculations (EA) take additional time, as outline in the EA table.
Transfers	The number of memory references. 4 clock cycles are required for each memory reference.
Bytes	Number of bytes in the instruction.

Note: The additional clock cycles required to reinitialize the instruction que and fetch the next instruction after a control transfer instruction (such as JMP or CALL) is already included in the timing tables. Two clock times are listed for conditional transfer instructions (such as JZ); the shortest time is for the case when there is no transfer.

Flags Register

The flags register contains various bits that control and record the state of the microprocessor, as defined below.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Flags Register:					OF	DF	IF	TF	SF	ZF		AF		PF		CF

Overflow	Set when an arithmetic overflow occurred. An arithmetic overflow occurs when the size of a computation exceeds the size of the destination.
Direction	Set for auto-decrement with string instructions, clear for auto-increment.
Interrupt Enable	Interrupts are enabled as long as this flag is set. When this flag is cleared, interrupts except for nonmaskable interrupts are disabled.
Trap	This flag is used by debuggers to single step through programs. When this flag is set, an IN T 3 is generated after every instruction.
Sign	Set when the high -order bit of the result is 1. In other words, S = 0 for positive numbers and S = 1 for negative numbers.
ZeroSet	Whenever the result is 0.
Auxiliary Carry	Set when the is a carry out of th e lower half of an 8 or 16 bit number, or when there is a borrow from the upper to the lower half.This flag is used mainly by the decimal-arithmetic instructions.
Parity Flag	Set if there is an even number of 1-bits in the result.Cleared if there is an odd number of 1-bits.Often used by communications programs
Carry Flag	Set if there was a carry out of, or a borrow into the high -order bit of the result.This flag is useful for propagating carries and borrows for multi-word numbers.

Effective Address (EA) Calculations

This table lists the number of clock cycles required to calculate the effective address on the 8088 microprocessor.The 80x8x processors require considerably fewer clock cycles to calculate effective addresses, so this table represents the worst-case.

EA Component	8088 Clocks	Example
Displacement	6	MOV AX,ADDR
Register indirect	5	MOV AX,[BP]
BX, BP, SI, DI		
Displacement + Base or Index	9	MOV AX,ADDR[BP]
BX + Disp, BP + Disp		
SI + Disp, DI + Disp		
Base + Index		
BP + DI, BX + SI	7	MOV AX,[BP+DI]
BP + SI, BX + DI	8	MOV AX,[BX+DI]
Displacement + Base + Index		
BP + DI + Disp	11	MOV AX,ADDR[BP+DI]
BX + SI + Disp		
BP + SI + Disp	12	MOV AX,ADDR[BP+SI]
BX + DI + Disp		

Note: Add 2 clocks for segment overrides.

Each memory reference requires an additional 4 clock cycles.The Transfers field in the instruction timing charts gives the number of memory references for each instruction.

Instruction Set

AAA ASCII Adjust after Addition

See also: AAD, AAS, AAM, ADC, DAA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	?

AAA

Logic:

```
If (AL & 0Fh) > 9 or (AF = 1) then
        AL ← AL + 6
        AH ←AH + 1
        AF ←1;CF ← 1
else
        AF ← 0;CF ← 0
        AL ← AL & 0Fh
```

Converts the number in the lower 4 bits (nibble) of AL to an unpacked BCD number (high-order nibble of AL is zeroed).

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	AAA

If the lower 4 bits of the number in AL is greater than 9, or the auxiliary carry flag is set, this instruction converts AL to its unpacked BCD form by adding 6 (subtra cting 10) to AL; adding 1 to AH; and setting the auxiliary flag and carry flags. This instruction will always leave 0 in the upper nibble of AL.

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least-significant digit.

AAD ASCII Adjust before Division

See Also: AAA, AAS, AAM, DIV, IDIV, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
?				*	*	?	*	?

AAD

Logic:

```
AL ← AH * 10 + AL
AH ← 0
```

AAD converts the unpacked two -digit BCD number in AX into binary in preparation for a division using DIV or IDIV, which require binary rather than BCD numbers.

Operands	Clocks	Transfers	Bytes	Example
no operands	60	-	2	AAD

AAD modifies the numerator in AL so that the result produced by a division will be a valid unpacked BCD numb er. For the subsequent DIV to produce the correct result, AH must be 0. After the division, the quotient is returned in AL, and the remainder in AH. Both high-order nibbles are zeroed.

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least-significant digit.

AAM ASCII Adjust after Multiply

See also: AAA, AAD, AAS, MUL, IMUL, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
?				*	*	?	*	?

AAM

Logic:

```
AH ← AL / 10
AL ← AL MOD 10
```

This instruction corrects the result of a previous multiplication of two valid unpacked BCD operands. A valid 2 -digit unpacked BCD number is taken from AX, the adjustment is performed, and the result is returned to AX. The high -order nibbles of the operands that were multiplied must have been 0 for this instruction to produce a correct result.

Operands	Clocks	Transfers	Bytes	Example
no operands	83	-	1	AAM

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least-significant digit.

AAS ASCII Adjust after Subtraction

See Also: AAA, AAD, AAS, SUB, SBB, DAS, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	?

AAS

Logic:

```
If (AL & 0Fh) > 9 or AF = 1 then
        AL ← AL - 6
        AH ← AH - 1
        AF ← 1;CF ← 1
else
        AF ← 0;CF ← 0
        AL ← AL & 0Fh
```

AAS corrects the re sult of a previous subtraction of two valid unpacked BCD operands, changing the content of AL to a valid BCD number. The destination operand of the subtraction must have been specified as AL. The high -order nibble of AL is always set to 0.

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	AAS

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least-significant digit.

ADC Add with Carry

See Also: ADD, INC, AAA, DAA, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

ADC destination,source

Logic: destination ← destination + source + CF

ADC adds the operands, adds 1 if the Carry Flag is set, and places the resulting sum in destination. Both operands may be bytes or words, and both may be signed or unsigned binary numbers.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, register	3	-	2	ADC BX,SI
register, immediate	4	-	3-4	ADC CX,128
accumulator, immediate	4	-	2-3	ADC AL,10
register, memory	9(13)+EA	1	2-4	ADC DX,RESULT
memory, register	16(24)+EA	2	2-4	ADC BETA,DI
memory, immediate	17(25)+EA	2	3-6	ADC GAMMA,16h

Note: ADC is useful for adding numbers that are larger than 16 bits, since it adds a carry from a previous operation.

ADD Addition

See Also: ADC, INC, AAA, DAA, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

ADD destination,source

Logic: destination ← destination + source

ADD sums the operands and stores the result in destination. Both operands may be bytes or words, and both may be signed or unsigned binary numbers.

Operands	Clocks	Transfers	Bytes	Example
	byte (word)			
register, register	3	-	2	ADD BX,CX
accumulator, immediate	4	-	2-3	ADD AX,256
register, immediate	4	-	3-4	ADD BL,4
register, memory	9(13)+EA	1	2-4	ADD DI,[DX]
memory, register	16(24)+EA	2	2-4	ADD TOTAL,BL
memory, immediate	17(25)+EA	2	3-6	ADD RESULT,3

AND Logical AND

See also: NOT, OR, XOR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

AND destination,source

Logic: destination ← destination AND source

AND performs a bit -by-bit logical AND operation on its operands and stores the result in destination. The operands may be words or bytes.

AND Instruction Logic:

Destination	Source	Result
0	0	0
0	1	0
1	0	0
1	1	1

AND sets each bit of the result to 1 if both of the corresponding bits of the operands are 1.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, register	3	-	2	AND AL,DL
register, immediate	4	-	3-4	AND CX,0FFh
accumulator, immediate	4	-	2-3	AND AX,01000010b
register, memory	9(13)+EA	1	2-4	AND CX,MASK
memory, register	16(24)+EA	2	2-4	AND VALUE,BL
memory, immediate	17(25)+EA	2	3-6	AND STATUS,01h

CALL Call Procedure

See also: RET, JMP, PROC, NEAR, FAR, EA

Flags not altered

CALL procedure_name

Logic:
```
if FAR CALL (inter-segment)
        PUSH CS
        CS ← dest_seg
PUSH IP
IP ← dest_offset
```

CALL transfers control to a procedure that can either be within the current segment (a NEAR procedure) or outside it (a FAR procedure). The two types of CALLs result in differe nt machine instructions, and the RET instruction that exits from the procedure must match the type of the CALL instruction (the potential for mismatch exists if the procedure and the CALL are assembled separately).

Operands	Clocks byte(word)	Transfers	Bytes	Example
near-proc	19(23)	1	3	CALL NEAR_PROC
far-proc	28(36)	2	5	CALL FAR_PROC
memptr 16	21(29)+EA	2	2-4	CALL PROC_TABLE[SI]
regptr 16	16(24)	1	2	CALL AX
memptr 32	37(57)+EA	4	2-4	CALL [BX].ROUTINE

Note: For an inter-segment procedure (procedure in a different segment), the processor first pushes the current value of CS onto the stack, then pushes the current value of IP (which is pointing to the instruction following the CALL instruction), then transfers control to the procedure.

For an intra-segment procedure (procedure in the same segment), the processor first pushes the current value of IP (which is pointing to the instruction following the CALL instruction) onto the stack, then transfers control to the procedure.

CALL can also read the p rocedure address from a register or memory location. This form of CALL is called an indirect CALL.

CBW Convert Byte to Word

See also: CWD, DIV, IDIV

Flags not altered

CBW

Logic:
```
if (AL < 80h) then
        AH ← 0
else
        AH ← FFh
```

CBW extends the sign bit of the AL register into the AH register. This instruction extends a signed byte value into the equivalent signed word value.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	CBW

Note: This instruction will set AH to 0FFh if the sign bit (bit 7) of AL is set; if bit 7 of AL is not set, AH will be set to 0. The instruction is useful for generating a word from a byte prior to performing byte division.

CLC Clear Carry Flag

See also: STC, CMC, STD, CLD, STI, CLI, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
								0

CLC

Logic: CF ← 0

CLC clears (sets to 0) the Carry Flag. No other flags are affected.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	CLC

CLD Clear Direction Flag

See also: STD, STC, CLC, CMC, STI, CLI, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
	0							

CLD

Logic: DF ← 0 (Increment in string instructions)

CLD zeros the Direction Flag. No other flags are affected. Clearing the direction flag causes string operations to increment SI and DI.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	CLD

Note: String instructions increment SI and DI when the direction flag is clear.

CLI Clear Interrupt-Enable Flag

See also: STI, STC, CLC, CMC, STD, CLD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
		0						

CLI

Logic: IF ← 0

CLI clears the Interrupt Enable Flag, suppressing processor recognition of maskable interrupts. No other flags are affected. (Non-maskable interrupts are recognized no matter what the state of the interrupt enable flag.)

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	CLI

CMC Complement Carry Flag

See also: STC, CLC, STD, CLD, STI, CLI, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
								*

CMC

Logic: CF ← CF

CMC reverses the current state of the Carry Flag.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	CMC

CMP Compare

See also: CMPS, SCAS, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

CMP destination,source

Logic: Flags set according to result of (destination - source)

CMP compares two numbers by subtracting the source from the destination and updates the flags. CMP does not change the source or destination. The operands may be bytes or words.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, register	3	-	2	CMP CX,BX
register, immediate	4	-	3-4	CMP BL,02h

accumulator, immediate	4	-	2-3	CMP AL,00010110b
register, memory	9(13)+EA	1	2-4	CMP DH,ALPHA_BETA
memory, register	9(13)+EA	1	2-4	CMP TOTAL,SI
memory, immediate	10(14)+EA	1	3-6	CMP VALUES,3420h

CMPS Compare String (Byte or Word)

See also: CMP, CMPSB, CMPSW, SCAS, REP, CLD, STD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

CMPS destination-string,source-string

Logic:

```
CMP (DS:SI), (ES:DI)        ; Sets flags only
if DF = 0
        SI ← SI + n         ; n = 1 for byte, 2 for word
        DI ← DI + n
else
        SI ← SI - n
        DI ← DI - n
```

This instruction compares two values by subtracting the byte or word pointed to by ES:DI, from the byte or word pointed to by DS:SI, and sets the flags according to the result s of the comparison. The operands themselves are not altered. After the comparison, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Operands	Clocks byte(word)	Transfers	Bytes	Example
dest,source	22(30)	2	1	CMPS STR1,STR2
(repeat) dest,source	9+22(30)/rep	2/rep	1	REPE CMPS STR1,STR2

Note: This instruction is always translated by the assembler into either CMPSB, Compare String Byt e, or CMPSW, Compare String Word, depending upon whether source refers to a string of bytes or words. In either case, you must explicitly load the SI and DI registers with the offset of the source and destination strings.

Example: Assuming the definition:

```
buffer1 db 100 dup (?)
buffer2 db 100 dup (?)
```

the following example compares BUFFER1 against BUFFER2 for the first mismatch.

```
          cld                           ;Scan in the forward direction
          mov cx, 100                   ;Scanning 100 bytes (CX is used by REPE)
          lea si, buffer1               ;Starting address of first buffer
          lea di, buffer2               ;Starting address of second buffer
repe      cmps buffer1,buffer2          ;           and compare it.
          jne mismatch                  ;The Zero Flag will be cleared if there
                                        ;is a mismatch
match:                  .               ;If we get here, buffers match
                        .
mismatch:
          dec si                        ;If we get here, we found a mismatch.
          dec di                        ;Back up SI and DI so they point to
                        .               ;the first mismatch
                        .
```

Upon exit from the REPE CMPS loop, the Zero Flag will be cleared if a mismatch was found, and set otherwise. If a mismatch was found, DI and SI will b e pointing one byte past the byte that didn't match; the DEC DI and DEC SI backup these registers so they point to the mismatched characters.

CMPSB Compare String Byte

See also: CMP, CMPS, CMPSW SCAS, REP, CLD, STD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

CMPSB

Logic:

```
CMP (DS:SI), (ES:DI)        ; Sets flags only
if DF = 0
        SI ← SI + 1
        DI ← DI + 1
else
        SI ← SI - 1
        DI ← DI - 1
```

This instruction compares two values by subtracting the byte pointed to by ES:DI, from the byte pointed to by DS:SI, and sets the flags according to the results of the comparison. The operands themselves a re not altered. After the comparison, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Operands	Clocks	Transfers	Bytes	Example
-	22	2	1	CMPSB
(repeat)	9+22/rep	2/rep	1	REPE CMPSB

Example: The following example compares BUFFER1 against BUFFER2 for the first mismatch.

```
          cld                          ;Scan in the forward direction
          mov cx, 100                  ;Scanning 100 bytes (CX is used by REPE)
          lea si, buffer1              ;Starting address of first buffer
          lea di, buffer2              Starting address of second buffer
repe      cmpsb                        ;...and compare it.
          jne mismatch                 ;The Zero Flag will be cleared if there
                                       ;is a mismatch
match:                  .              ;If we get here, buffers match
                        .
mismatch:
          dec si                       ;If we get here, we found a mismatch.
          dec di                       ;Back up SI and DI so they point to the
                        .              ;first mismatch
```

Upon exit from the REPE CMPSB loop, the Zero Flag will be cleared if a mismatch was found, and set otherwise. If a mismatch was found, DI and SI will be poin ting one byte past the byte that didn't match; the DEC DI and DEC SI instructions backup these registers so they point to the mismatched characters.

CMPSW Compare String Word

See Also: CMP, CMPS, CMPSB, SCAS, REP, CLD, STD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

CMPSW

```
Logic:  CMP (DS:SI), (ES:DI)     ; Sets flags only
        if DF = 0
                SI ← SI + 2
                DI ← DI + 2
        else
                SI ← SI - 2
                DI ← DI - 2
```

This instruction compares two numbers by subtracting the word pointedto by ES:DI, from the word pointed to by DS:SI, and sets the flags according to the results of the comparison. The operands themselves a re not altered. After the comparison, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flagis set), in preparation for comparing the next element of the string.

Operands	Clocks	Transfers	Bytes	Example
-	30	21	CMPSW	
(repeat)	9 + 30/rep	2/rep	1	REPE CMPSW

Example: The following example compares BUFFER1 against BUFFER2 for the first mismatch.

```
          cld                 ;Scan in the forward direction
          mov cx, 50          ;Scanning 50 words (100 bytes)
          lea si, buffer1     ;Starting address of first buffer
          lea di, buffer2     ;Starting address of second buffer
repe      cmps                ;...and compare it.
          jne mismatch        ;The Zero Flag will be cleared if there
                              ;is a mismatch
match:             .          ;If we get here, buffers match
                   .
mismatch:
          dec si              ;If we get here, we found a mismatch.
          dec si              ;Back up DI and SI so they point to the
          dec di              ;…first mismatch
          dec di
```

Upon exit from the REPE CMPSW loop, the Zero Flag will be cleared if a mismatch was found, and set otherwise. If a mismatch was found , DI and SI will be pointing one word (two bytes) past the word that didn't match; the DEC DI and DEC SI pairs backup these registers so they point to the mismatched characters.

CWD Convert Word to Doubleword

See Also: CBW, DIV, IDIV

Flags: not altered

CWD

```
Logic:          if (AX < 8000h) then
                        DX ← 0
                else
```

```
DX ← FFFFh
```

CWD extends the sign bit of the AX register into the DX register. This instruction generates the double -word equivalent of the signed number in the AX register.

Operands	Clocks	Transfers	Bytes	Example
no operands	5	-	1	CWD

Note: This instruction will set DX to 0FFFFh if the sign bit (bit 15) of AX is set; if bit 15 of AX is not set, DX will be set to 0.

DAA Decimal Adjust after Addition

See Also: DAS, AAA, AAS, AAM, AAD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
?				*	*	*	*	*

DAA

Logic:

```
If (AL & 0Fh) > 9 or (AF = 1) then
          AL ← AL + 6
          AF ← 1
else
          AF ← 0
If (AL > 9Fh) or (CF = 1) then
          AL ← AL + 60h
          CF ← 1
else
          CF ← 0
```

DAA corrects the result of a previous addition of two valid packed decimal operands (note that this result must be in AL). This instruction changes the content of AL so that it will contain a pair of valid packed decimal digits.

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	DAA

Note: Packed BCD stores one digit per nibble (4 bits); the least significant digit is in the lower nibble. It is not possib le to apply an adjustment after division or multiplication of packed BCD numbers. If you need to use multiplication or division, it is better to use unpacked BCD numbers.See, for example, the description of AAM (ASCII Adjust after Multiply).

DAS Decimal Adjust after Subtraction

See Also: AAS, DAA, SUB, SBB, DEC, NEG, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
?				*	*	*	*	*

DAS

Logic:

```
If(AL & 0Fh) > 9 or (AF = 1) then
          AL ← AL - 6
          AF ← 1
else
          AF ← 0
If (AL > 9Fh) or (CF = 1) then
          AL ← AL - 60h
          CF ← 1
else
          CF ← 0
```

DAS corrects the result of a previous subtraction of two valid packed decimal operands (note that this result must be in AL). This instruction changes the content of AL so that it will contain a pair of valid packed decimal digits.

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	DAS

Note: Packed BCD stores one digit per nibble (4 bits); the least significant digit is in the lower nibble.

It is not possible to apply an adjustment after division or multiplication of packed BCD numbers. If you need to use multiplication and division, it is better to use unpacked BCD numbers. See, for example, the description of AAM (ASCII Adjust after Multiply).

DEC Decrement

See Also: SUB, SBB, AAS, DAS, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

DEC destination

Logic: destination ← destination - 1

This instruction decrements the destination by one. The destination operand, which may be either a word or a byte, is treated as an unsigned binary number.

Operands	Clocks	Transfers	Bytes	Example
reg16	2	-	1	DEC BX
reg8	3	-	2	DEC BL
memory	15(23)+EA	2	2-4	DEC MATRIX[SI]

Note: This instruction does not set the carry, so if you need to decrement a multi -word number, it is better to use the SUB and SBB instructions.

DIV Divide, Unsigned

See Also: IDIV, SHR, AAD, CBW, CWD, INT 00h, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

DIV source

```
Logic:      AL ← AX / source          ;Source is byte
            AH ← remainder
    or
            AX ← DX:AX / source       ;Source is word
            DX ← remainder
```

This instruction performs unsigned division. If the source is a byte, DIV divides the word value in AX by source, returning the quotient in AL and the rem ainder in AH. If the source is a word, DIV divides the double -word value in DX:AX by the source, returning the quotient in AX and the remainder in DX.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**
reg8	0-90	-	2	DIV BL
reg16	4-162	-	2	DIV BX
mem8	(86-96)+EA	1	2-4	DIV VYUP
mem16	(154-172)+EA	1	2-4	DIV NCONQUER[SI]

Note: If the result is too large to fit in the destination (AL or AX), an INT 0 (Divide by Zero) is generated, and the quotient and remainder are undefined.

When an Interrupt 0 (Divide by Zero) is generated, the saved CS:IP value on the 80286 and 80386 points to the instruction that failed (the DIV instruction).On the 8088/8086, however, CS:IP points to the instruction following the failed DIV instruction.

ESC Escape

See Also: HLT, WAIT, LOCK, EA

Flags: not altered

ESC coprocessor's-opcode, source

ESC is used to pass control from the microprocessor to a coprocessor, such as an 8087 or 80287. In response to ESC, the microprocessor accesses a memory operand -the instruction for the coprocessor -and pl aces it on the bus. The coprocessor watches for ESC commands and executes the instruction placed on the bus, using the effective address of source.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**
	byte(word)			
immediate, memory	8(12)+ EA	1	2-4	ESC 6,ADR[SI]
immediate, register	2	-	2	ESC COPROC-CODE,AH

Note: In order to synchronize with the math coprocessor, the programmer must explicitly code the WAIT instruction preceding all ESC instructions. The 80286 and 80386 have automatic instruction synchronization, hence WAITs are not needed.

HLT Halt the Processor

See also: WAIT, ESC, LOCK

Flags: not altered

HLT

This instruction halts the CPU. The processor leaves the halted state in response to a non -maskable interrupt; a maskable interrupt with interrupts enabled; or activation of the reset line.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**
no operands	2	-	1	HLT

IDIV Integer Divide, Signed

See also: DIV, SAR, AAD, CBW, CWD, INT 00h, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

IDIV source

```
Logic:      AL ← AX / source          ; Byte source
            AH ← remainder
```

or
AX 🡐 DX:AX / source ; Word source
DX 🡐 remainder

IDIV performs signed division. If source is a byte, IDIV divides the word value in AX by source, returning the quotient in AL and the remainder in AH. If source is a word, IDIV divides the double-word value in DX:AX by source, returning the quotient in AX and the remainder in DX.

Operands	Clocks	Transfers	Bytes	Example
reg8	101-112	-	2	IDIV CL
reg16	165-184	-	2	IDIV DX
mem8	(107-118)+EA	1	2-4	IDIV BYTE[SI]
mem16	(175-194)+EA	1	2-4	IDIV [BX].WORD_ARRAY

Note: If the result is too large to fit in the destination (AL or AX), an INT 0 (Divide by Zero) is generated, and the quotient and remainder are undefined.

The 80286 and 80386 microprocessors are able to generate the largest negative number (80h or 8000h) as a quotient for this instruction, but the 8088/8086 will generate an Interrupt 0 (Divide by Zero) if this situation occurs.

When an Interrupt 0 (Divide by Zero) is generated, the saved CS:IP value on the 80286 and 80386 points to the instruction that failed (the IDIV instruction). On the 8088/8086, however, CS:IP points to the instruction following the failed IDIV instruction.

IMUL — Integer Multiply, Signed

See Also: MUL, AAM, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

IMUL source

Logic: AX🡐 AL * source ; if source is a byte
or
DX:AX 🡐 AX * source ; if source is a word

IMUL performs signed multiplication. If source is a byte, IMUL multiplies source by AL, returning the produc t in AX. If source is a word, IMUL multiplies source by AX, returning the product in DX:AX. The Carry Flag and Overflow Flag are set if the upper half of the result (AH for a byte source, DX for a word source) contains any significant digits of the product , otherwise they are cleared.

Operands	Clocks	Transfers	Bytes	Example
reg8	80-98	-	2	IMUL CL
reg16	128-154	-	2	IMUL BX
mem8	(86-104)+EA	1	2-4	IMUL BITE
mem16	(138-164)+EA	1	2-4	IMUL WORD[BP][DI]

IN — Input Byte or Word

See Also: OUT

Flags: not altered

IN accumulator,port

Logic: accumulator 🡐 (port)

IN transfers a byte or a word from a port to AL or AX. The port may be specified by an immediate byte value (for ports 0 through 255) or by the DX register (allowing access to all ports).

Operands	Clocks byte(word)	Transfers	Bytes	Example
accumulator, immed8	10(14)	1	2	IN AL,45h
accumulator, DX	8(12)	1	1	IN AX,DX

Note: It is advised that hardware not use I/O ports F8h through FFh, since these are reserved for controlling the math coprocessor and future processor extensions.

INC — Increment

See Also: ADD, ADC, AAA, DAA, DEC, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

INC destination

Logic: destination 🡐 destination + 1

INC adds 1 to the destination. The destination, which may be either a byte or a word, is considered an unsigned binary number.

Operands byte(word)	Clocks	Transfers	Bytes	Example
reg16	2	-	1	INC BX

reg8	3	-	2	INC BL
memory	15(23)+EA	2	2-4	INC THETA[BX]

Note: This instruction does not set the carry flag.If you need to add 1 to a multi -word number, it is better to use the ADD and ADC instructions instead.

INT Interrupt

See also: IRET INTO PUSHF CALL PUSHF INT 03h Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
0	0							

INT interrupt-num

Logic:

```
PUSHF              ;Push flags onto stack
TF ← 0             ;Clear Trap Flag
IF ← 0             ;Disable Interrupts
CALL FAR (INT*4)   ;Call the interrupt handler
```

INT pushes the flags register, clears the Trap and Interrupt-enable Flags, pushes CS and IP, then transfers control to the interrupt handler specified by the interrupt-num.If the interrupt handler returns using an IRET instruction, the original flags are restored.

Operands	Clocks byte(word)	Transfers	Bytes	Example
immed8 (type=3)	52	5	1	INT 3
immed8 (type<>3)	51	5	2	INT 21

Note: The flags are store d in the same format as that used by the PUSHF instruction. The address of the interrupt vector is determined by multiplying the interrupt-num by 4. The first word at the resulting address is loaded into IP, and the second word into CS.

All interrupt -nums except type 3 generate a two -byte opcode; type 3 generates a one -byte instruction called the Breakpoint interrupt.

INTO Interrupt on Overflow

See Also: INT, IRET, JNO, JO, PUSHF, CALL, INT 04h, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
0	0							

INTO

Logic:

```
if (OF = 1)
          PUSHF              ;Push flags onto stack
          TF ← 0             ;Clear Trap Flag
IF ← 0                       ;Disable Interrupts
          CALL FAR (10h)     ;The INTO vector is at 0000:0010h
```

INTO activates interrupt type 4 if the Overflow Flag is set; otherwise it does nothing. This interrupt operates identically to an "INT 4" if the overflow flag is set, in which case INTO pushes the flags register, clears the Trap and Interrupt -enable Flags, pushes CS and IP, then transfers control to the interrupt -num 4 handler, which is pointed to by the vector at location 10h.If the interrupt handler returns using an IRET instruction, the original flags are restored.

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

Note: The flags are stored in the same format as that used by the PUSHF instruction. INTO can be used after an operation that may cause overflow, to call a recovery procedure.

IRET Interrupt Return

See Also: INT, INTO, POP, POPF

Flags Affected:

O	D	I	T	S	Z	A	P	C
r	r	r	r	r	r	r	r	r

IRET

Logic:

```
POP IP
POP CS
POPF          ; Pop flags from stack
```

IRET returns control from an interrupt routine to the point where the interrupt occurred, by popping IP, CS, and the Flag registers.

Operands	Clocks	Transfers	Bytes	Example
no operands	32	3	1	IRET

JA Jump If Above

See Also: JNBE, JAE, JG, JBE

Flags: not altered

JA short-label

Jump Condition: Jump if CF = 0 and ZF = 0

Used after a CMP or SUB instruction, JA transfers control to short - label if the first operand (which should be unsigned) was greater than the second operand (also unsigned). The target of the jump mus t be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JA ABOVE

Note: JNBE, Jump Not Below or Equal, is the same instruction as JA.

JA, Jump on Above, should be used when comparing unsigned numbers.

JG, Jump on Greater, should be used when comparing signed numbers.

JAE Jump If Above or Equal

See Also: JNB, JA, JGE, JB

Flags: not altered

JAE short-label

Jump Condition: Jump if CF = 0

Used after a CMP or SUB instruction, JAE transfers control to short- label if the first operand (which should be unsigned) was greater than or equal to the second operand (also unsigned). The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JAE ABOVE_EQUAL

Note: JNB (Jump Not Below) is the same instruction as JAE.

JAE, Jump on Above or Equal, should be used whencomparing unsigned numbers.

JGE, Jump on Greater or Equal, should be used when comparing signed numbers.

JB Jump If Below

See Also: JNAE, JC, JL, JAE

Flags: not altered

JB short-label

Jump Condition: Jump if CF = 1

Used after a CMP or SUB instruction, JB transfers control to short - label if the first operand was less than the second.(Both operands are treated as unsigned numbers.)The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JB BELOW

Note: JC (Jump if Carry), JB, and JNAE (Jump if Not Above or Equal), are all synonyms for the same instruction.
JB, Jump if Below, should be used when comparing unsigned numbers.

JL, Jump if Less Than, should be used when comparing signed numbers.

JBE Jump If Below or Equal

See Also: JNA, JLE, JA

Flags: not altered

JBE short-label

Jump Condition: Jump if CF = 1 or ZF = 1

Used after a CMP or SUB instruction, JBE transfers control to short - label if the first operand was less than or equal to the second. (Both operands are treated as unsigned numbers.)The target of the ju mp must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JBE NOT_ABOVE

Note: JNA, Jump if Not Above, is the same instruction as JBE.

JBE, Jump if Below or Equal, should be used when comparing unsigned numbers.

JLE, Jump if Less Than or Equal, should be used when comparing signed numbers.

JC Jump If Carry

See Also: JB, JNAE, JNC

Flags: not altered

JC short-label

Jump Condition: Jump if CF = 1

JC transfers control to short -label if the Carry Flag is set. The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JC CARRY_SET

Note: JB (Jump if Below), JC, and JNAE (Jump if Not Above or Equal) are all synonyms for the same instruction.

Use JNC, Jump if No Carry, to jump if the carry flag is clear.

JCXZ Jump If CX Register Zero

See Also: LOOP, LOOPE, LOOPZ, LOOPNZ, LOOPNE

Flags: not altered

JCXZ short-label

Jump Condition: Jump if CX = 0

JCXZ transfers control to short -label if the CX register is 0. The target of the jump must be within -128 to +127 bytes of the nextinstruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	18 or 6	-	2	JCXZ COUNT_DONE

Note: This instruction is commonly used at the beginning of a loop to bypass the loop if the counter variable (CX) is at 0.

JE Jump If Equal

See Also: JZ, JNE

Flags: not altered

JE short-label

Jump Condition: Jump if ZF = 1

Used after a CMP or SUB instruction, JE transfers control to s hort- label if the first operand is equal to the second operand. The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JE ZERO

Note: JZ, Jump if Zero, is the same instruction as JE

JG Jump If Greater

See also: JNLE, JA, JLE

Flags: not altered

JG short-label

Jump Condition: Jump if ZF = 0 and SF = OF

Used after a CMP or SUB instruction, JG transfers control to short - label if the first operand is greater than t he second. (Both operands are treated as signed numbers.)The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JG GREATER

Note: JNLE, Jump if Not Less or Equal, is the same instruction as JG.

JA, Jump if Above, should be used when comparing unsigned numbers.

JG, Jump if Greater, should be used when comparing signed numbers.

JGE Jump If Greater or EQual

See also: JNL, JAE, JL

Flags: not altered

JGE short-label

Jump Condition: Jump if SF = OF

Used after a CMP or SUB instruction, JGE transfers control to short - label if the first operand is greater than or equal to the second. (Both operands are treated as signed numbers.)The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JGE GREATER_EQUAL

Note: JNL, Jump if Not Less, is the same instruction as JGE.

JAE, Jump if Above or Equal, should be used when comparing unsigned numbers.

JGE, Jump if Greater or Equal, should be used when comparing signed numbers.

JL Jump If Less

See Also: JNGE, JB, JGE

Flags: not altered

JL short-label

Jump Condition: Jump if SF <> OF

Used after a CMP or SUB instruction, JL transfers control to short - label if the first operand is less than the second. (Both operands are treated as signed numbers.)The target of the jump must be within - 128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JL LESS

Note: JNGE, Jump if Not Greater or Equal, is the same instruction as JL.

JB, Jump if Below, should be used when comparing unsigned numbers.

JL, Jump if Less, should be used when comparing signed numbers.

JLE Jump If Less or Equal

See Also: JNG, JBE, JNA, JG

Flags: not altered

JLE short-label

Jump Condition: Jump if SF <> OF or ZF = 1

Used after a CMP or SUB instruction, JLE transfers control to short- label if the first operand is less than or equal to the second . (Both operands are treated as signed numbers.)The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JLE NOT_GREATER

Note: JNG, Jump if Not Greater, is the same instruction as JLE.

JBE, Jump if Below or Equal, should be used when comparing unsigned numbers.

JLE, Jump if Less or Equal, should be used when comparing signed numbers.

JMP Jump Unconditionally

See also:CALL, RET, SHORT, NEAR, FAR, PROC, EA

Flags: not altered

JMP target

Jump Condition: Jump always

JMP always transfer control to the target location. Unlike CALL, JMP does not save IP, because no RETurn is expected. An intrasegment JMP may be made either through memory or through a 16 -bit register; an intersegment JMP can be ma de only through memory.

Operands	Clocks	Transfers	Bytes	Example
short-label	15	-	2	JMP ROPE_NEAR
near-label	15	-	3	JMP SAME_SEGMENT
far-label	15	-	5	JMP FAR_LABEL
memptr16	18+EA	-	2-4	JMP SAME_SEG
regptr16	11	-	2	JMP BX
memptr32	24+EA	-	2-4	JMP NEXT_SEG

Note: If the assembler can determine that the target of an intrasegment jump is within 127 bytes of the current location, the assembler will automatically generate a short-jump (two-byte) instruction; otherwise, a 3- byte NEAR JMP is generated.

You can force the generation of a short jump by explicit use of the operator "short," as in:

```
JMPshort near_by
```

JNA Jump If Not Above

See Also: JBE, JLE

Flags: not altered

JNA short-label

JNA is a synonym for JBE. See the description for JBE.

JNAE Jump If Not Above or Equal

See Also: JB, JL

Flags: not altered

JNAE short-label

JNAE is a synonym for JB. See the description for JB.

JNB Jump If Not Below

See Also: JAE, JGE

Flags: not altered

JNB short-label

JNB is a synonym for JAE. See the description for JAE.

JNBE Jump If Not Below or Equal

See Also: JA, JG

Flags: not altered

JNBE short-label

JNBE is a synonym for JA. See the description for JA.

JNC Jump If No Carry

See Also: JC

Flags: not altered

JNC short-label

Jump Condition: Jump if CF = 0

JNC transfers contr ol to short -label if the Carry Flag is clear. The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JNC CARRY_CLEAR

Note: Use JC, Jump if Carry, to jump if the carry flag is set.

JNE Jump If Not Equal

See Also: JNZ, JE

Flags: not altered

JNE short-label

Jump Condition: Jump if ZF = 0

Used after a CMP or SUB instruction, JNE transfers control to short - label if the first operand is not equal to the second. The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JNE NOT_EQUAL

Note: JNZ, Jump if Not Zero, is the same instruction as JNE.

JNG Jump If Not Greater

See Also: JLE, JBE

Flags: not altered

JNG short-label

JNG is a synonym for JLE. See the description for JLE.

JNGE Jump If Not Greater or Equal

See Also: JLJB

Flags: not altered

JNGE short-label

JNGE is a synonym for JL. See the description for JL.

JNL Jump If Not Less

See Also: JGE, JAE

Flags: not altered

JNL short-label

JNL is a synonym for JGE. See the description for JGE.

JNLE Jump If Not Less or Equal

See Also: JG, JA

Flags: not altered

JNLE short-label

JNLE is a synonym for JG. See the description for JG.

JNO Jump If No Overflow

See Also: JO, INTO

Flags: not altered

JNO short-label

Jump Condition: Jump if OF = 0

JNO transfers control to short-label if the Overflow Flag is clear. The target of the jump must be within -128 to +127 bytes of the next instruction.

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

Note: Use JO, Jump if Overflow, to jump if the overflow flag is set.

JNP Jump If No Parity

See Also: JP, OJP

Flags: not altered

JNP short-label

Jump Condition: Jump if PF = 0

JNP transfers control to short -label if the Parity Flag is clear. The target of the jump must be within -128 or +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JNP ODD_PARITY

Note: JPO, Jump if Parity Odd, is the same instruction a JNP.

Use JP, Jump on Parity, to jump if the parity flag is set.

JNS Jump If No Sign

See Also: JS

Flags: not altered

JNS short-label

Jump Condition:Jump if SF = 0

JNS transfers control to short-label if the Sign Flag is clear. The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JNS AQUARIUS

Note: Use JS, Jump if Sign, to jump if the sign flag is set.

JNZ Jump If Not Zero

See Also: JNE, JE

Flags: not altered

JNZ short-label

JNZ is a synonym for JNE. See the description for JNE.

JO Jump If Overflow

See Also: JNO, INTO

Flags: not altered

JO short-label

Jump Condition: Jump if OF = 1

JO transfers control to short -label if the Overflow Flag is set.The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JO SIGNED_OVERFLOW

Note: Use JNO, Jump if No Overflow, to jump if the overflow flag is clear.

JP Jump If Parity

See Also: JPE, JNP

Flags: not altered

JP short-label

Jump Condition: Jump if PF = 1

JP transfers control to short -label if the Parity Flag is set. The target of the jump must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	16 or 4	-	2	JP EVEN_PARITY

Note: JPE, Jump if Parity Even, is the same instruction as JP.

Use JNP, Jump if No Parity, to jump if the Parity Flag is clear.

JPE Jump If Parity Even

See Also: JP, JNP

Flags: not altered

JPE short-label

JPE is a synonym for JP. See the description for JP.

JPO Jump If Parity Odd

See also: JNP, JP

Flags: not altered

JPO short-label

JPO is a synonym for JNP. See the description for JNP.

JS Jump If Sign

See also: JNS

Flags: not altered

JS short-label

Jump Condition: Jump if SF = 1

JS transfers control to short -label if the Sign Flag is set. The target of the jump must be within -128 to +127 bytes of the next instruction.

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

Note: Use JNS, Jump if No Sign, to jump if the sign flag is clear.

JZ Jump If Zero

See also: JE, JNE

Flags: not altered

JZ short-label

JZ is a synonym for JE. See the description for JE.

LAHF Load Register AH from Flags

See also: SAHF, PUSHF, POPF

Flags: not altered

LAHF

Logic: AH bits←Flag-reg bits
7 6 4 2 0 ←S Z A P C

LAHF copies the five 8080/8085 flags (Sign, Zero, Auxiliary Carry, Parity, and Carry) into bits 7, 6, 4, 2, and 0, respectively, of the AH register. The flags themselves are unchanged by this instruction.

Operands	Clocks	Transfers	Bytes	Example
no operands	4	-	1	LAHF

Note: This instruction is primarily used to provide upward compatibility between the 8080/8085 family and the 8086 family.

After this instruction is executed, bits 1, 3 and 5 of AH are undefined.

LDS — Load Pointer using DS

See also: LEA, LES, OFFSET, EA

Flags: not altered

LDS destination,source

Logic: DS ← (source + 2)
destination ← (source)

LDS loads into two registers the 32 -bit pointer variable found in memory at source. LDS stores the segment value (the higher order word of source) in DS and the offset value (the lower order word of source) in the des tination register. The destination register may be any any 16-bit general register (that is, all registers except segment registers).

Operands	Clocks	Transfers	Bytes	Example
reg16, mem32	24+EA	2	2-4	LDS DI,32_POINTER

Note: LES, Load Pointer Using ES, is a comparable instruction that loads the ES register rather than the DS register.

LEA — Load Effective Address

See Also: LDS, LES, OFFSET, EA

Flags: not altered

LEA destination,source

Logic: destination ← Addr(source)

LEA transfers the offset of the source operand (rather than its value) to the destination operand. The source must be a memory reference, and the destination must be a 16-bit general register.

Operands	Clocks	Transfers	Bytes	Example
reg16, mem16	2+EA	-	2-4	LEA BX,MEM_ADDR

Note: This instruction has an advantage over using the OFFSET operator with the MOV instruction, in that the source operand can be subscripted. For example, this is legal:

```
LEA BX, TABLE[SI]                ;Legal
```

whereas

```
MOV BX, OFFSET TABLE[SI]         ;Not legal
```

is illegal, since the OFFSET operator performs its calculation at assembly time and this address is not known until run time.

Example: The DOS print string routine, Function 09h, requires a pointer to the string to be printed in DS:DX. If the string you wished to print was at address "PRINT -ME" in the same data segment, you could load DS:DX with the required pointer using this instruction:

```
LEA DX, PRINT-ME
```

LES — Load Pointer using ES

See Also: LEA, LDS, OFFSET, EA

Flags: not altered

LES dest-reg,source

Logic: ES ← (source)
dest-reg ← (source + 2)

LES loads into two registers the 32 -bit pointer variable found in memory at source. LES stores the segment value (the higher order word of source) in ES and the offset value (the lower order word of source) in the destin ation register. The destination register may be any any 16-bit general register (that is, all registers except segment registers).

Operands	Clocks	Transfers	Bytes	Example
reg16, mem32	24+EA	2	2-4	LES DI, STR_ADDR

Note: LDS, Load Pointer Using DS, is a comparable instruction that loads the DS register rather than the ES register.

LOCK — Lock the Bus

See Also: HLT, WAIT, ESC

Flags: not altered

LOCK

LOCK is a one -byte prefix that can precede any instruction. LOCK causes the processor to assert its bus lock signal while the instruction that follows is executed. If the system is configured such that the LOCK signal is used, it prevents any external device or event from accessing the bus, including interrupts and DMA transfers.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	LOCK XCHG FLAG,AL

Note: This instruction was provided to support multiple processor systems with shared resources. In such a system, access to those resources is generally controlled via a software-hardware combination using semaphores.
This instruction should only be used to prevent other bus masters from interrupting a data movement operation. This prefix should only be used with XCHG, MOV, and MOVS.

LODS — Load String (Byte or Word)

See also: LODSB, LODSW, CMPS, MOVS, SCAS, STOS, REP, CLD

Flags: not altered

LODS source-str

Logic:

```
Accumulator ← (DS:SI)
if DF = 0
    SI ← SI + n ; n = 1 for byte, 2 for word scan
else
    SI ← SI - n
```

LODS transfers the value (word or byte) pointed to by DS:SI into AX or AL.It also increments or decrements SI (depending on the state of the Direction Flag) to point to the next element.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**
	byte(word)			
source-str	12(16)	-	1	LODS LIST
(repeat) source-str	9+13(17)/rep	1/rep	1	REP LODS LIST

Note: This instruction is always translated by the assembler into either LODSB, Load String B yte, or LODSW, Load String Word, depending upon whether source-str refers to a string of bytes or words. In either case, however, you must explicitly load the SI register with the offset of the string.

Although it is legal to repeat this instruction, it i s almost never done since doing so would continually overwrite the value in AL.

Example: The following example sends the eight bytes at INIT_PORT to port 250. (Don't try this on your machine, unless you know what's hanging off port 250.)

```
INIT_PORT:
        DB      '$CMD0000'              ;The string we want to send
                    .
                    .
        CLD                             ;Move forward through string at INIT_PORT
        LEA SI,INIT_PORT                ;SI gets starting address of string
        MOV CX,8                        ;CX is counter for LOOP instruction
AGAIN:
        LODS INIT_PORT                  ;"INIT_PORT" is needed only by the
        OUT 250,AL                      ;assembler, for determining word or byte
        LOOP AGAIN
```

LODSB — Load String Byte

See Also: LODS, LODSW, CMPS, MOVS, SCAS, STOS, REP, CLD, STD

Flags: not altered

LODSB

Logic:

```
AL ← (DS:SI)
if DF = 0
    SI ← SI + 1
else
    SI ← SI - 1
```

LODSB transfers the byte pointed to by DS:SI into AL and increments or decrements SI (depending on the state of the Direction Flag) to point to the next byte of the string.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**
-	12	-	1	LODSB
(repeat)	9+13/rep	1/rep	1	REP LODSB

Note: Although it is legal to repeat this instruction, it is almost never done since doing so would continually overwrite the value in AL.

Example: The following example sends the eight bytes at INIT_PORT to port 250. (Don't try this on your machine, unless yo u know what's hanging off port 250.)

```
INIT_PORT:
        DB '$CMD0000'           ;The string we want to send
              .
              .
        CLD                     ;Move forward through string at INIT_PORT
        LEA SI, INIT_PORT       ;SI gets starting address of string
        MOV CX, 8               ;CX is counter for LOOP instruction
AGAIN:
        LODSB                   ;Load a byte into AL...
        OUT 250,AL              ; ...and output it to the port.
        LOOP AGAIN
```

LODSW Load String Word

See also: LODS, LODSB, CMPS, MOVS, SCAS, STOS, REP, CLD, STD

Flags: not altered

LODSW

Logic:

```
AX ← (DS:SI)
if DF = 0
        SI ← SI + 2
else
        SI ← SI - 2
```

LODSW transfers the word pointed to by DS:SI into AX and increments or decrements SI (depending on the state of the Direction Flag) to point to the next word of the string.

Operands	Clocks	Transfers	Bytes	Example
-	16	-	1	LODSW
(repeat)	9+17/rep	1/rep	1	REP LODSW

Note: Although it is legal to repeat this instruction, it is almost never done since doing so would continually overwrite the value in AL.

Example: The following example sends the eight bytes at INIT_PORT to port 250. (Don't try this on your machine, unless you know what's hanging off port 250.)

```
INIT_PORT:
        DB'$CMD0000'                    ;The string we want to send
                    .
                    .
        CLD                             ;Move forward through string at INIT_PORT
        LEA SI, INIT_PORT               ;SI gets starting address of string
        MOV CX, 4                       ;Moving 4 words (8 bytes)
AGAIN:
        LODSW                           ;Load a word into AX...
        OUT 250,AX                      ; ...and output it to the port.
        LOOP AGAIN
```

LOOP Loop on Count

See Also: LOOPE, LOOPNE, LOOPNZ, LOOPZ, JCXZ

Flags: not altered

LOOP short-label

Logic:

```
CX ← CX - 1
If (CX <> 0)
        JMP short-label
```

LOOP decrements CX by 1, then transfers control to short -label if CX is not 0.Short -label must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	17/5	-	2	LOOP AGAIN

LOOPE Loop While Equal

See also: LOOP, LOOPNE, LOOPNZ, LOOPZ, JCXZ

Flags: not altered

LOOPE short-label

Logic:

```
CX ← CX - 1
If CX <> 0 and ZF = 1
        JMP short-label
```

Used after a CMP or SUB, LOOPE decrements CX by 1, then transfers control to short -label if the first operand of the CMP or SUB is equal to the second operand. Short-label must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	18 or 6	-	2	LOOPE AGAIN

Note: LOOPZ, Loop if Zero, is the same instruction.

LOOPNE Loop While not Equal

See also: LOOPZ, LOOP, LOOPE, LOOPNZ, JCXZ

Flags: not altered

LOOPNE short-label

Logic:

```
CX ← CX - 1
If CX <> 0 and ZF = 0
    JMP short-label
```

Used after a CMP or SUB, LOOPNE decrements CX by 1, then transfers control to short-label if the first operand of the CMP or SUB is not equal to the second operand. Short-label must be within -128 to +127 bytes of the next instruction.

Operands	Clocks	Transfers	Bytes	Example
short-label	19 or 5	-	2	LOOPNE AGAIN

Note: LOOPNZ, Loop While Not Zero, is the same instruction.

LOOPNZ Loop While not Zero

See Also: LOOPNE, LOOP, LOOPE, JCXZ

Flags: not altered

LOOPNZ short-label

LOOPNZ is a synonym for LOOPNE. See the description for LOOPNE.

LOOPZ Loop While Zero

See also: LOOPE, LOOP, LOOPNE, JCXZ

Flags: not altered

LOOPZ short-label

LOOPZ is a synonym for LOOPE. See the description for LOOPE.

MOV Move (Byte or Word)

See also: MOVSPUSHPOPXCHGXLATEA

Flags: not altered

MOV destination,source

Logic: destination ← source

MOV copies a byte or word from the source into the destination.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, register	2	-	2	MOV BX,CX
memory, accumulator	10(14)	1	3	MOV MEM_DEST,AL
accumulator, memory	10(14)	1	3	MOV AX,MEM_SOURCE
memory, register	9(13)+EA	1	2-4	MOV MEM_DEST,BX
register, memory	8(12)+EA	1	2-4	MOV BX,MEM_SOURCE
register, immediate	4	-	2-3	MOV BX,0F6CDh
memory, immediate	10(14)+EA	1	3-6	MOV MASK,0F6CDh
seg-reg, reg16	2	-	2	MOV DS,BX
seg-reg, mem16	8(12)+EA	1	2-4	MOV DS,SEGMENT_VAL
reg16, seg-reg	2	-	2	MOV BP,SS
memory, seg-reg	9(13)+EA	1	2-4	MOV CODE_VAR,CS

MOVS Move String (Byte or Word)

See Also: MOV, MOVSB, MOVSW, CMPS, LODS, SCAS, STOS, REP, CLD, STD

Flags: not altered

MOVS destination-string,source-string

Logic:

```
(ES:DI) ← (DS:SI)
if DF = 0
    SI ← SI + n                ; n = 1 for byte, 2 for word scan
    DI ← DI + n
else
    SI ← SI - n
    DI ← DI - n
```

This instruction copies the byte or word pointed to by DS:SI, into the location pointed to by ES:DI. After the move, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), to point to the next element of the string.

Operands	Clocks byte(word)	Transfers	Bytes	Example
dest,source	18(26)	2	1	MOVS WORD_BUFF,INPUT
(repeat) dest,source	9+17(25)/rep	2/rep	1	REP MOVSW

Note: This instruction is always translated by the assembler into either MOVSB, Move String By te; or MOVSW, Move String Word, depending upon whether source -string refers to a string of bytes or words. In either case, you must explicitly load the SI and DI registers with the offset of the source and destination strings.

Example: Assuming BUFFER1 as been defined somewhere as:

```
BUFFFER1DB100 DUP (?)
```

the following example moves 100 bytes from BUFFER1 to BUFFER2:

```
CLD                              ;Move in the forward direction
LEA SI, BUFFER1                  ;Source address to SI
LEA DI, BUFFER2                  ;Destination address to DI
MOV CX,100                       ;CX is used by the REP prefix
REP MOVSBUFFER2, BUFFER1         ;...and MOVS it.
```

MOVSB — Move String Byte

See Also: MOV, MOVS, MOVSW, CMPS, LODS, SCAS, STOS, REP, CLD, STD

Flags: not altered

MOVSB

Logic:

```
(ES:DI) ← (DS:SI)
if DF = 0
        SI ← SI + 1
        DI ← DI + 1
else
        SI ← SI - 1
        DI ← DI - 1
```

This instruction copies the byte pointed to by DS:SI into the location pointed to by ES:DI. After the move, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), to point to the next byte.

Operands	Clocks	Transfers	Bytes	Example
-	18	2	1	MOVSB
(repeat)	9+17/rep	2/rep	1	REP MOVSB

Example: Assuming BUFFER1 as been defined somewhere as:

```
BUFFFER1DB100 DUP (?)
```

the following example moves 100 bytes from BUFFER1 to BUFFER2:

```
CLD                        ;Move in the forward direction
LEA SI, BUFFER1            ;Source address to SI
LEA DI, BUFFER2            ;Destination address to DI
MOV CX,100                 ;CX is used by the REP prefix
REP MOVSB                  ;...and move it.
```

MOVSW — Move String Word

See also: MOV, MOVS, MOVSB, CMPS, LODS, SCAS, STOS, REP, CLD, STD

Flags: not altered

MOVSW

Logic:

```
(ES:DI) ← (DS:SI)
if DF = 0
        SI ← SI + 2
        DI ← DI + 2
else
        SI ← SI - 2
        DI ← DI - 2
```

This instruction copies the word pointed to by DS:SI to the location pointed to by ES:DI. After the move, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), to point to the next word.

Operands	Clocks	Transfers	Bytes	Example
-	26	2	1	MOVSW
(repeat)	9+25/rep	2/rep	1	REP MOVSW

Example: Assuming BUFFER1 as been defined somewhere as:

```
BUFFFER1DB100 DUP (?)
```

the following example moves 50 words (100 bytes) from BUFFER1 to BUFFER2:

```
CLD                     ;Move in the forward direction
LEA SI, BUFFER1         ;Source address to SI
LEA DI, BUFFER2         ;Destination address to DI
MOV CX,50               ;Used by REP; moving 50 words
REP MOVSW               ;...and move it.
```

MUL — Multiply, Unsigned

See Also: IMUL, AAM, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

MUL source

Logic: AX ← source * AL ;if source is a byte
or
DX:AX = source * AX ;if source is a word

MUL performs unsigned multiplication. If source is a byte, MUL multiplies source by AL, returning the product in AX. If source is a word, MUL multiplies source by AX, returning the product in DX:AX. The Carry and Overflow flags are set if the upper half of the result (AH for a byte source, DX for a word source) contains any significant digits of the product, otherwise they are cleared.

Operands	Clocks	Transfers	Bytes	Example
reg8	70-77	-	2	MUL CH
reg16	118-133	-	2	MUL BX
mem8	(76-83)+EA	1	2-4	MUL A_BYTE
mem16	(128-143)+EA	1	2-4	MUL A_WORD

NEG — Negate

See also: NOT, SUB, SBB, AAS, DAS, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

NEG destination

Logic: destination ← -destination ; two's complement

NEG subtracts t he destination operand from 0 and returns the result in the destination. This effectively produces the two's complement of the operand. The operand may be a byte or a word.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register	3	-	2	NEG DL
memory	16(24)+EA	2	2-4	NEG COEFFICIENT

Note: If the operand is zero, the carry flag is cleared; in all other cases, the carry flag is set.

Attempting to negate a byte containing -128 or a word containing -32,768 causes no change to the operand and sets t he Overflow Flag.

NOP — No Operation

Flags: not altered

Logic: None

NOP causes the processor to do nothing. This instruction is frequently used for timing purposes, to force memory alignment, and as a "place- holder."

Operands	Clocks	Transfers	Bytes	Example
no operands	3	-	1	NOP

NOT — Logical NOT

See Also: NEG, AND, OR, XOR, EA

Flags: not altered

NOT destination

Logic: destination ← NOT(destination) ; one's complement

NOT inverts each bit of its operand (that is, forms the one's complement). The operand can be a word or byte.

NOT Instruction Logic

Destination	Result
0	1
1	0

Operands	Clocks byte(word)	Transfers	Bytes	Example
register	3	-	2	NOT DX
memory	16(24)+EA	2	2-4	NOT MASK

OR Logical OR

See Also: AND, NOT, XOR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

OR destination,source

Logic: destination ← destination OR source

OR performs a bit -by-bit logical inclusive OR operation on its operands and returns the result to destination.The operands may be words or bytes.

OR Instruction Logic

Destination	Source	Result
0	0	0
0	1	1
1	0	1
1	1	1

OR sets each bit of the result to 1 if either or both of the corresponding bits of the operands are 1.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, register	3	-	2	OR CH,DL
register, memory	9(13)+EA	1	2-4	OR BX, MEM_OR_Y
memory, register	16(24)+EA	2	2-4	OR MEM_OR_Y, BX
accumulator, immediate	4	-	2-3	OR AL,01110110b
register, immediate	4	-	3-4	OR CX,00FFh
memory, immediate	17(25)+EA	2	3-6	OR MEM_WORD,76h

OUT Output to Port

See Also: IN

Flags: not altered

OUT port,accumulator

Logic: (port) ← accumulator

OUT transfers a byte or a word from AL or AX to the specified port. The port may be specified with an immediate byte constant (allowing access to ports 0 through 255) or with a word value in DX (allowing access to ports 0 through 65,535).

Operands	Clocks	Transfers	Bytes	Example
byte(word)				
immed8, accumulator	10(14)	1	2	OUT 254,AX
DX, accumulator	8(12)	1	1	OUT DX,AL

Note: It is advised that hardware not use I/O ports F8h through FFh, since these are reserved for controlling the math coprocessor and future processor extensions.

POP POP a Word from the Stack

See Also: PUSH, POPF, MOV, XCHG, XLAT, EA

Flags: not altered

POP destination

Logic: destination ← (SP)
SP ← SP + 2

POP transfers the word at the top of the stack to the destination operand, then increments SP by 2 to poi nt to the new top of stack.

Operands	Clocks	Transfers	Bytes	Example
register	12	1	1	POP CX
seg-reg (CS illegal)	12	1	1	POP ES
memory	25+EA	2	2-4	POP VALUE

Note: You may not use the CS register as the destination of a POP instruction.

POPF POP Flags from the Stack

See also: POP, PUSH, PUSHF, LAHF, SAHF

Flags Affected:

O	D	I	T	S	Z	A	P	C
r	r	r	r	r	r	r	r	r

Logic: flag-register ← (SP)
SP ← SP + 2

POPF transfers the word at the top of the stack to the flags register, replacing the old flags, then increments SP by 2 to point to the new top of stack.

Operands	Clocks	Transfers	Bytes	Example
no operands	12	1	1	POPF

PUSH Push Word onto Stack

See also:POP, POPF, PUSHF, MOV, XCHG, EA

Flags: not altered

PUSH source

Logic: SP ← SP - 2
(SP) ← source

PUSH decrements SP by 2, then copies the operand to the new top of stack. The source of a PUSH instruction cannot be an 8 -bit register.

Operands	Clocks	Transfers	Bytes	Example
register	15	1	1	PUSH BX
seg-reg (CS illegal)	14	1	1	PUSH ES
memory	24+EA	2	2-4	PUSH PARAMETERS

Note: Even if the source refers to a byte in memory, a full word is always pushed.

The 80286 and 80386 microprocess ors will push a different value on the stack for the instruction PUSH SP than will the 8086/8088.The 80286 and 80386 push the value of SP before SP is incremented, while the 8086/8088 increments SP first, then pushes SP on the stack.Use the following code instead of a PUSH SP in order to obtain the same results on all microprocessors.

```
PUSH BP
MOV BP, SP
XCHGBP, [BP]
```

This code functions in the same manner as a PUSH SP on the 8088/8086.

PUSHF Push Flags onto Stack

See Also: POPF, LAHF, SAHF

Flags: not altered

Logic: SP ← SP - 2
(SP) ← flag-register

PUSHF decrements SP by 2, then transfers the flags register to the new top of stack.

Operands	Clocks	Transfers	Bytes	Example
no operands	14	1	1	PUSHF

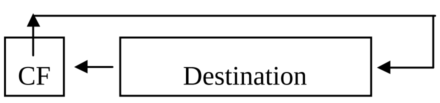
RCL Rotate through Carry Left

See Also: ROL, ROR, RCR, SHL, SHR, SAR, SAL, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*								*

RCL destination,count



RCL shifts the word or byte at the destination to the left by the number of bit positions specified in the second operand, COUNT. A bit shifted out of the left (high-order) end of the destination enters the carry flag, and the displaced carry flag rotates around to enter the vacated right-most bit position of the destination. This "bit rotation" continues the number of times specified in COUNT. (Another way of looking at this is to consider the carry flag as the highest order bit of the word being rotated.)

If COUNT is not equal to 1, the Overflow flag is undefined. If COUNT is equal to 1, then the Overflow Flag is set to the XOR of the top 2 bits of the original operand.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, 1	2	-	2	RCL CX,1

register, CL	8+4/bit	-	2	RCL BL,CL
memory, 1	15(23)+EA	2	2-4	RCL MULTIPLY_X_2,1
memory, CL	20(28)+EA+4/bit	2	2-4	RCL MOVE_AROUND,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to rotate only one position, replace the second operand, CL, with the value 1, as shown in the first example above.

The 80286 and 80386 microprocessors limit the COUNT value to 31.If the C OUNT is greater than 31, these microprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time that an interrupt response will be delayed waiting for the instruction to complete.

Multiple RCLs t hat use 1 as the COUNT may be faster and require less memory than a single RCL that uses CL for COUNT.

The overflow flag is undefined if the rotate count is greater than 1.

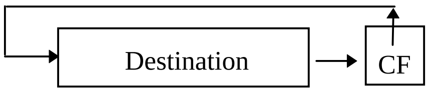
RCR Rotate through Carry Right

See Also: ROR, RCL, ROL, SAR, SHR, SHL, SAL, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*								*

RCR destination, count



RCR shifts the word or byte at the destination to the right by the number of bit positions specified in the second operand, COUNT. A bit shifted out of the right (low-order) end of the destination enters the carry flag, and the displaced carry flag rotates around to enter the vacated left -most bit position of the destination. This "bit rotation" continues the number of times specified in COUNT. (Another way of looking at this is to consider the carry flag as the lowest order bit of the word being rotated.)
If COUNT is not equal to 1, the Overflow flag is undefined. If COUNT is equal to 1, the Overflow Flag is set to the XOR of the top 2 bits of the result.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, 1	2	-	1	RCR CX,1
register, CL	8+4/bit	-	2	RCR DL,CL
memory, 1	15(23)+EA	2	2-4	RCR DIVIDE_BY_2,1
memory, CL	20(28)+EA+4/bit	2	2-4	RCR AROUND_MOVE,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to rotate only one position, replace the second operand, CL, with the value 1, as shown in the first example above.

The 80286 and 80386 microprocessors limit the COUNT value to 31.If COUNT is greater than 31, these m icroprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time an interrupt response will be delayed waiting for the instruction to complete.

Multiple RCRs that use 1 as the COUNT may be faster and require less memory than a single RCR that uses CL for COUNT.

The overflow flag is undefined if the rotate count is greater than 1.

REP Repeat

See Also: REPNE, MOVS, STOS, CMPS, SCAS, LODS, CLD, STD

Flags: not altered

REP string-instruction

Logic:

```
while CX <> 0                          ;for MOVS, LODS or STOS
        execute string instruction
        CX ← CX - 1
------------------------------------
while CX <> 0
        execute string instruction     ;for CMPS or SCAS
        CX ← CX - 1
        if ZF = 0 terminate loop
```

REP is a prefix that may be specified before any string instruction (C MPS, LODS, MOVS, SCAS, and STOS). REP causes the string instruction following it to be repeated, as long as CX does not equal 0; CX is decremented after each execution of the string instruction.(For CMPS and SCAS, REP will also terminate the loop if the Ze ro Flag is clear after the string instruction executes.)

Operands	Clocks	Transfers	Bytes	Example
-	2	-	1	REP MOVS TO,FROM

Note: If CX is initially 0, the REPeated instruction is skipped entirely.

The test for CX equal to 0 is performed before the ins truction is executed.The test for the Zero Flag clear --done only for CMPS and SCAS--is performed after the instruction is executed.

REP, REPE (Repeat While Equal), and REPZ (Repeat While Zero) are all synonyms for the same instruction.

REPNZ (Repeat Not Zero) is similar to REP, but when used with CMPS and SCAS, will terminate with the Zero Flag set, rather than cleared.

REP is generally used with the MOVS (Move String) and STOS (Store String) instructions; it can be thought of as "repeat while not end of string."

You do not need to initialize ZF before using repeated string instructions.

A REPeated instruction that is interrupted between repeats will correctly resume processing upon return from the interrupt. However, if other prefixes are used on a single instruction (for example, segment override or LOCK) in addition to the REP, all prefixes except the one that immediately preceded the string instruction will be lost. Therefore, if you must use more than one prefix on a single instruction, you should d isable interrupts before the instruction, and enable them afterwards. Note that even this precaution will not prevent a non -maskable interrupt, and that lengthy string operations may cause large delays in interrupt processing.

Example: The following example moves 100 bytes from BUFFER1 to BUFFER2:

```
CLD                          ;Move in the forward direction
LEA SI, BUFFER1              ;Source pointer to SI
LEA DI, BUFFER2              ;...and destination to DI
MOV CX,100                   ;REP uses CX as the counter
REP MOVSB                    ;...and do it
```

REPE Repeat While Equal

See Also: REP

Flags: not altered

This instruction is a synonym for REP. See the description for REP.

REPNE Repeat While not Equal

See Also: REP, MOVS, STOS, CMPS, SCAS, LODS, CLD, STD

Flags: not altered

REPNE string-instruction

Logic:

```
while CX <> 0                          ;for MOVS, LODS or STOS
      execute string instruction
      CX ← CX - 1
-----------------------------------
while CX <> 0                          ;for CMPS or SCAS
      execute string instruction
      CX ← CX - 1
      if ZF <> 0 terminate loop        ;This is only difference
                                       ;between REP and REPNE
```

REPNE is a prefix that may be specified before any string instruction (CMPS, LODS, MOV S, SCAS, and STOS). REPNE causes the string instruction following it to be repeated, as long as CX does not equal 0; CX is decremented after each execution of the string instruction. (For CMPS and SCAS, REP will also terminate the loop if the Zero Flag is set after the string instruction executes. Compare this to REP, which will terminate if the Zero Flag is clear.)

Operands	Clocks	Transfers	Bytes	Example
-	2	-	1	REPNE SCASB

Note: If CX is initially 0, the REPeated instruction is skipped entirely.

The test for CX equal to 0 is performed before the instruction is executed.The test for the Zero Flag set --done only for CMPS and SCAS--is performed after the instruction is executed.

REPNE and REPNZ are synonyms for the same instruction.

You do not need to initialize ZF before using repeated string instructions.

A repeated instruction that is interrupted between repeats will correctly resume processing upon return from the interrupt. However, if other prefixes are used on a single instruction (for example, segment override or LOCK) in addition to the REP, all prefixes except the one that immediately preceded the string instruction will be lost. Therefore, if you must use more than one prefix on a single instruction, you should disable interrupts before the i nstruction, and enable them afterwards. Note that even this precaution will not prevent a non -maskable interrupt, and that lengthy string operations may cause large delays in interrupt processing.

Example: The following example will find the first byte equal to 'A' in the 100-byte buffer at STRING.

```
CLD                          ;Scan string in forward direction
MOV AL,'A'                   ;Scan for 'A'
LEA DI, STRING               ;Address to start scanning at
```

```
MOV CX,100                ;Scanning 100 bytes
REPNE SCASB               ; ...and scan it
DEC DI                    ;Back up DI to point to the 'A'
```

Upon termination of the repeated SCASB instruction, CX will be equal to zero if a byte value of 'A' was not found in STRING, and non-zero if it was.

REPNZ — Repeat While Not Zero

See Also: REPNE

Flags: not altered

REPNZ is the same instruction as REPNE. See the description for REPNE.

REPZ — Repeat While Zero

See Also: REP

Flags: not altered

This instruction is a synonym for REP. See the description for REP.

RET — Return from Procedure

See Also: IRET, CALL, JMP

Flags: not altered

RET optional-pop-value

Logic:

```
POP IP
If FAR RETURN (inter-segment)
        POP CS
        SP ← SP + optional-pop-value (if specified)
```

RET transfers control from a called procedure back to the instruction following the CALL, by:

- Popping the word at the top of the stack into IP
- If the return is an intersegment return:
  - Popping the word now at the top of the stack into CS
- Adding the optional-pop-value, if specified, to SP

The assembler will generate an intrasegment RET if the procedure containing the RET was designated by the programmer as NEAR, and an intersegment RET if it was designated FAR. The o ptional-pop-value specifies a value to be added to SP, which has the effect of popping the specified number of bytes from the top of the stack.

Operands	Clocks	Transfers	Bytes	Example
(intrasegment, no pop)	20	1	1	RET
(intrasegment, with pop)	24	1	3	RET 4
(intersegment, no pop)	32	2	1	RET
(intersegment, pop)	31	2	3	RET 2

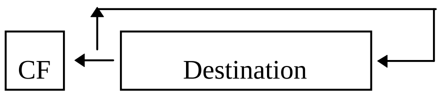
ROL — Rotate Left

See Also: RCL, ROR, RCR, SAL, SAR, SHL, SHR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*								*

ROL destination,count



ROL shifts the word or byte at the destination to the left by the number of bit positions specified in the second operand, COUNT. As bits are transferred out the left (high -order) end of the destination, they re -enter on the right (low -order) end. The Carry Flag is updated to match the last bit shifted out of the left end.
If COUNT is not equal to 1, the Overflow flag is undefined. If COUNT is equal to 1, then the Overflow Flag is set to the XOR of the top 2 bits of the original operand.

Operands	Clocks	Transfers	Bytes	Example
byte(word)				
register, 1	2	-	2	ROL DI,1
register, CL	8+4/bit	-	2	ROL BX,CL
memory, 1	15(23)+EA	2	2-4	ROL BYTE,1
memory, CL	20(28)+EA+4/bit	2	2-4	ROL WORD,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to ro tate only one position, replace the second operand, CL, with the value 1, as shown in the first example above.

The 80286 and 80386 microprocessors limit the COUNT value to 31.If COUNT is greater than 31, these microprocessors will use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper boundary exists to limit the amount of time that an interrupt response will be delayed waiting for the instruction to complete.

Multiple ROLs that use 1 as the COUNT may be faster and require less memory than a single ROL that uses CL for COUNT.

The overflow flag is undefined when the rotate count is greater than 1.

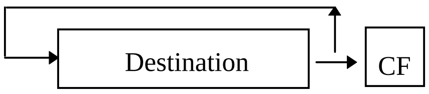
ROR Rotate Right

See Also: RCR, ROL, RCL, SAR, SAL, SHL, SHR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*								*

ROR destination,count



ROR shifts the word or byte at the destination to the right by the number of bit positions specified in the second operand, COUNT. As bits are transferred out the right (low -order) end of the destination, they re -enter on the left (high-order) end. The Carry Flag is updated to match the last bit shifted out of the right end.

If COUNT is not equal to 1, the Overflow flag is undefined. If COUNT is equal to 1, the Overflow Flag is set to the XOR of the top 2 bits of the result.

Operands	**Clocks** byte(word)	**Transfers**	**Bytes**	**Example**
register, 1	2	-	2	ROR BL,1
register, CL	8+4/bit	-	2	ROR AX,CL
memory, 1	15(23)+EA	2	2-4	ROR WORD,1
memory, CL	20(28)+EA+4/bit	2	2-4	ROR BYTE,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to rotate by only one position, replace the second operand, CL, with the value 1, as shown in the first example above.

The 80286 and 80386 microprocessors limit the COUNT value to 31.If the COUNT is greater than 31, these microprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time an interrupt response will be delayed waiting for the instruction to complete.

Multiple RORs that use 1 as the COUNT may be faster and require le ss memory than a single ROR that uses CL for COUNT.

The overflow flag is undefined when the rotate count is greater than 1.

SAHF Store Register AH into Flags

See Also: LAHF, PUSHF, POPF, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
				*	*	*	*	*

Logic: Flag-reg bits ← AH bits
S Z A P C ←7 6 4 2 0

SAHF copies bits 7, 6, 4, 2, and 0 from the AH register into the Flags register, replacing the previous values of the Sign, Zero, Auxiliary Carry, Parity, and the Carry flags.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**
no operands	4	-	1	SAHF

Note: The Overflow, Direction, Interrupt, and Trap flags are not changed by this instruction. This instruction is primarily used to provide upward compatibility between the 8080/8085 family and the 8086 family.

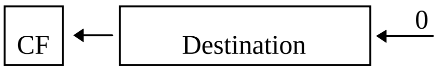
SAL Shift Arithmetic Left

See also: SHL, SAR, SHR, RCL, RCR, ROL, ROR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

SAL destination,count



SAL shifts the word or byte at the destination to the left by the number of bit positions specified in the second operand, COUNT. As bits are transferred out the left (high -order) end of the destination, zeroes are shifted in the right (low -order) end. The Carry Flag is updated to match the last bit shifted out of the left end. If COUNT is not equa l to 1, the Overflow flag is undefined. If COUNT is equal to 1, the Overflow Flag is cleared if the top 2 bits of the original operand were the same, otherwise the Overflow Flag is set.

Operands	**Clocks**	**Transfers**	**Bytes**	**Example**

	byte(word)			
register, 1	2	-	2	SAL AL,1
register, CL	8+4/bit	-	2	SAL SI,CL
memory, 1	15(23)+EA	2	2-4	SAL WORD,1
memory, CL	20(28)+EA+4/bit	2	2-4	SAL BYTE,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to shift only one position, replace the second operand, CL, with the value 1, as shown in the first example above.

The 80286 and 80386 microprocessors limit the COUNT value to 31.If the COUNT is greater than 31, these microprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time an interrupt response will be delayed waiting for the instruction to complete.

Multiple SALs that use 1 as the COUNT may be faster and require less memory than a single SAL that uses CL for COUNT.

SHL, Shift Logical Left, is the same instruction; SHL is a synonym for SAL.

The overflow flag is undefined when the shift count is greater than 1.

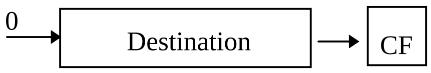
SAR Shift Arithmetic Right

See Also: SHR, SAL, SHL, RCR, RCL, ROR, ROL, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

SAR destination,count



SAR shifts the word or byte in destination to the right by the number of bit positions specified in the second operand, COUNT. As bits are transferred out the right (low -order) end of the destin ation, bits equal to the original sign bit are shifted into the left (high-order) end, thereby preserving the sign bit. The Carry Flag is set equal to the last bit shifted out of the right end. If COUNT is not equal to 1, the Overflow flag is undefined. If COUNT is equal to 1, the Overflow flag is cleared.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, 1	2	-	2	SAR DX,1
register, CL	8+4/bit	-	2	SAR DI,CL
memory, 1	15(23)+EA	2	2-4	SAR N_BLOCKS,1
memory, CL	20(28)+EA+4/bit	2	2-4	SAR N_BLOCKS,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to shift only one position, replace the second operand, CL, with the value 1, as shown in the first example above.

The 80286 and 80386 microprocessors limit the COUNT valu e to 31.If the COUNT is greater than 31, these microprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time an interrupt response will be delayed waiting for the instruction to complete.

Multiple SARs that use 1 as the COUNT may be faster and require less memory than a single SAR that uses CL for COUNT.

The overflow flag is undefined when the shift count is greater than 1.

SBB Subtract with Borrow

See Also: SUB, DEC, NEG, AAS, DAS, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

SBB destination,source

Logic: destination ← destination - source - CF

SBB subtracts the source from the destination, subtracts 1 from that result if the Carry Flag is set, and stores the result in destination. The operands may be bytes or words, and both may be signed or unsigned binary numbers.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, register	3	-	2	SBB DX,AX
register, memory	9(13)+EA	1	2-4	SBB DX,FEE
memory, register	16(24)+EA	2	2-4	SBB SIGH,SI
accumulator, immediate	4	-	2-3	SBB AX,8
register, immediate	4	-	3-4	SBB BH,4
memory, immediate	17(25)+EA	2	3-6	SBB TOTAL,10

Note: SBB is useful for subtracting numbers that are larger than 16 bits, since it subtracts a borrow (in the carry flag) from a previous operation.

You may subtract a byte -length immediate value from a destination which is a word; in this case, the byte is sign - extended to 16 bits before the subtraction.

SCAS Scan String (Byte or Word)

See Also: SCASB, SCASW, CMP, CMPS, REP, CLD, STD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

SCAS destination-string

Logic:

```
CMP Accumulator, (ES:DI);Sets flags only
if DF = 0
          DI ← DI + n ;n = 1 for byte, 2 for word
else
          DI ← DI - n
```

This instruction compares the accumulator (AL or AX) to the byte or word pointed to by ES:DI. SCAS sets the flags according to the results of the comparison; the operands themselves are not altered. Af ter the comparison, DI is incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Operands	Clocks byte(word)	Transfers	Bytes	Example
dest-str	15(19)	1	1	SCAS WORD_TABLE
(repeat) dest-str	9+15(19)/rep	1/rep	1	REPNE SCAS BYTE_TABLE

Note: This instruction is always translated by the assembler into either SCASB, Scan String Byte, or SCASW, Scan String Word, depending upon whether destination -string refers to a string of bytes or words. In either case, however, you must explicitly load the DI register with the offset of the string.

SCAS is useful for searching a block for a given byte or word value. Use CMPS, Compare String, if you wish to compare two strings (or blocks) in memory, element by element.

Example: Assuming the definition:

```
LOST_A DB 100DUP(?)
```

the following example searches the 100-byte block starting at LOST_A for the character 'A' (65 decimal).

```
          MOV AX, DS
          MOV ES, AX                        ;SCAS uses ES:DI, so copy DS to ES
          CLD                               ;Scan in the forward direction
          MOV AL, 'A'                       ;Searching for the lost 'A'
          MOV CX,100                        ;Scanning 100 bytes (CX is used by REPNE)
          LEA DI, LOST_A                    ;Starting address to DI
          REPNE SCASLOST_A                  ;...and scan it.
          JE FOUND                          ;The Zero Flag will be set if we found a match.
NOTFOUND:                     .             ;If we get here, no match was found
                              .
                              .
FOUND:         DEC DI                       ;Back up DI so it points to the first
                              .             ; matching 'A'
                              .
```

Upon exit from the REPNE SCAS loop, the Zero Flag will be set if a match was found, and cle ared otherwise. If a match was found, DI will be pointing one byte past the match location; the DEC DI at FOUND takes care of this problem.

SCASB Scan String Byte

See Also: SCAS, SCASW, CMPS, REP, CLD, STD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

SCASB

Logic:

```
CMP AL, (ES:DI)          ; Sets flags only
if DF = 0
          DI ← DI + 1
else
          DI ← DI - 1
```

This instruction compares two bytes by subtracting the destination byte, pointed to by ES:DI, from AL. SCASB sets the flags according to the results of the comparison. The operands themselves are not altered. After the comparison , DI is incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next byte.

Operands	Clocks	Transfers	Bytes	Example
-	15	1	1	SCASB
(repeat)	9+15/rep	1/rep	1	REPNE SCASB

Note: SCAS is useful for searching a block for a given byte or word value. Use CMPS, Compare String, if you wish to compare two strings (or blocks) in memory, element by element.

Example: The following example searches the 100-byte block starting at LOST_A for the character 'A' (65 decimal).

```
          MOV AX, DS
          MOV ES, AX                         ;SCAS uses ES:DI, so copy DS to ES
          CLD                                ;Scan in the forward direction
          MOV AL, 'A'                        ;Searching for the lost 'A'
          MOV CX,100                         ;Scanning 100 bytes (CX is used by REPNE)
          LEA DI, LOST_A                     ;Starting address to DI
          REPNE SCASB                        ; ...and scan it.
          JE FOUND                           ;The Zero Flag will be set if we found a match.
     NOTFOUND:                 .             ;If we get here, no match was found
                               .
                               .
     FOUND:     DEC DI                       ;Back up DI so it points to the first
                               .             ;matching 'A'
                               .
```

Upon exit from the REPNE SCASB loop, the Zero Flag will be set if a match was found, and cleared otherwise. If a match was found, DI will be pointing one byte past the match location; the DEC DI at FOUND takes care of this problem.

SCASW — Scan String Word

See Also: SCAS, SCASB, CMPS, REP, CLD, STD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

SCASW

Logic:

```
CMP AX, (ES:DI) ; Sets flags only
if DF = 0
          DI ← DI + 2
else
          DI ← DI - 2
```

This instruction compares two words by subtracting the destination word, pointed to by ES:DI, from AX. SCASW sets the flags according to the results of the comparison. The operands themselves are not altered. After the comparison , DI is incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next word.

Operands	Clocks	Transfers	Bytes	Example
-	19	1	1	SCASW
(repeat)	9+19/rep	1/rep	1	REPNE SCASW

Note: SCAS is useful for searching a block for a given byte or word value. Use CMPS, Compare String, if you wish to compare two strings (or blocks) in memory, element by element.

Example: The following example searches the 100-byte block starting at LOST_A for the character 'A' (65 decimal).

```
          MOV AX, DS
          MOV ES, AX                         ;SCAS uses ES:DI, so copy DS to ES
          CLD                                ;Scan in the forward direction
          MOV AL, 'A'                        ;Searching for the lost 'A'
          MOV CX,50                          ;Scanning 50 words (CX is used by REPNE)
          LEA DI, LOST_A                     ;Starting address to DI
          REPNE SCASW                        ; ...and scan it.
          JE FOUND                           ;The Zero Flag will be set if we found a match.
          REPNE SCASW                        ; ...and scan it.
          JE FOUND                           ;The Zero Flag will be set if we found a match.
     NOTFOUND:                 .             ;If we get here, no match was found
                               .
                               .
     FOUND: DEC DI                           ;Back up DI so it points to the first
          DEC DI                             ;matching 'A'
                               .
                               .
```

Upon exit from the REPNE SCASW loop, the Zero Flag will be set if a match was found, and cleared otherwise. If a match was found, DI will be pointing two bytes (one word) past th e match location; the DEC DI pair at FOUND takes care of this problem.

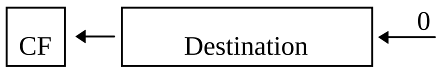
SHL — Shift Logical Left

See Also: SAL, SHR, SAR, RCL, RCR, ROL, ROR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

SHL destination,count



SHL is the same instruction as SAL, Shift Arithmetic Left. SHL shifts the word or byte at the destination to the left by the number of bit positions specified in the second operand, COUNT. As bits are transferred out the left (high -order) end of the destination, zeroes are shifted in the right (low -order) end. The Carry Flag is updated to match the last bit shifted out of the left end.

If COUNT is not equal to 1, the Overflow flag is undefined. If COUNT is equal to 1, the Overflow Flag is cleared if the top 2 bits of the original operand were the same, otherwise the Overflow Flag is set.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
register, 1	2	-	2	SHL AL,1
register, CL	8+4/bit	-	2	SHL SI,CL
memory, 1	15(23)+EA	2	2-4	SHL WORD,1
memory, CL	20(28)+EA+4/bit	2	2-4	SHL BYTE,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to shift only one position, replace the second operand, CL, with the value 1, as shown in the first example below.

The 80286 and 80386 microprocessors limit the COUNT value to 31.If the COUNT is greater than 31, these microprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time an interrupt response will be delayed waiting for the instruction to complete.

Multiple SHLs that use 1 as the COUNT may be faster and require less memory than a single SHL that uses CL for COUNT.

The overflow flag is undefined when the shift count is greater than 1.

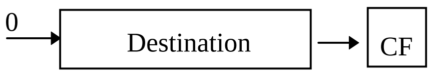
SHR Shift Logical Right

See Also: SAR, SHL, SAL, RCR, RCL, ROR, ROL, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

SHR destination,count



SAR shifts the bits in destination to the right by the number of positions specified in the count operand (or in CL, if n o count operand is included). 0s are shifted in on the left. If the sign bit retains its original value, the Overflow Flag is cleared; it is set if the sign changes. The Carry Flag is updated to reflect the last bit shifted.

If COUNT is not equal to 1, th e Overflow flag is undefined, otherwise the Overflow Flag is set to the high -order bit of the original operand.

Operands	Clocks	Transfers	Bytes	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],
memory, CL	20+EA+4/bit	2	2-4	SHR INPUT_WORD,CL

Note: COUNT is normally taken as the value in CL. If, however, you wish to shift only one position, replace the second operand, CL, with the value 1, as shown in the first example below.

The 80286 an d 80386 microprocessors limit the COUNT value to 31.If the COUNT is greater than 31, these microprocessors use COUNT MOD 32 to produce a new COUNT between 0 and 31.This upper bound exists to limit the amount of time an interrupt response will be delayed waiting for the instruction to complete.

Multiple SHRs that use 1 as the COUNT may be faster and require less memory than a single SHR that uses CL for COUNT.

The overflow flag is undefined when the shift count is greater than 1.

STC Set Carry Flag

See Also: CLC, CMC, STD, CLD, STI, CLI, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
								1

Logic: CF $\leftarrow$ 1

STC sets the Carry Flag. No other flags are affected.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	STC

STD Set Direction Flag

See Also: CLD, STC, CLC, CMC, STI, CLI, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
	1							

Logic: DF 🡐 1 (Decrement in string instructions)

STD sets the Direction Flag. No other flags are affected. Setting the direction flag causes string operations to decreme nt SI and DI.

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	STD

STI Set Interrupt Enable Flag

See Also: CLI, STC, CLC, CMC, STD, CLD, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
		1						

Logic: IF 🡐 1

STI sets the Interrupt Enable Flag, permitting the processor to recognize maskable interrupts. No other flags are affected. (Non - maskable interrupts are recognized no matter what the state of the interrupt enable flag.)

Operands	Clocks	Transfers	Bytes	Example
no operands	2	-	1	STI

Note: A pending interrupt will not be recognized until after the instruction following the STI executes.

STOS Store String (Byte or Word)

See Also: STOSB, STOSW, CMPS, LODS, MOVS, SCAS, REP, CLD, STD

Flags: not altered

STOS destination-string

Logic:

```
(ES:DI) 🡐 Accumulator
if DF = 0
        DI 🡐 DI + n          ; n = 1 for byte, 2 for word scan
else
        DI 🡐 DI - n
```

STOS copies the value (byte or word) in AL or AX into the location pointed to by ES:DI. DI is then incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing the accumulator in the next location.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
dest-string	11(15)	1	1	STOS WORD_ARRAY
(repeat) dest-string	9+10(14)/rep	1/rep	1	REP STOS BYTE_ARRAY

Note: This instruction is always translated by the assembler into either STOSB, Store String Byte, or STOSW, Store String Word, depending upon whether destination -string refers to a string of bytes or words. In either case, however, you must explicitly load the DI register with the offset of the string.

Example: When used in conjunction with the REP prefixes, the Store String instructions are useful for initializing a block of memory. For example, the following code would initialize the 100-byte memory block at BUFFER to 0:

```
MOV AL,0                 ;The value to initialize BUFFER to
LEA DI,BUFFER            ;Starting location of BUFFER
MOV CX,100               ;Size of BUFFER
CLD                      ;Let's move in forward direction
REP STOS BUFFER          ;Compare this line to example for STOSB
```

STOSB Store String Byte

See Also: STOS, STOSW, CMPS, LODS, MOVS, SCAS, REP, CLD, STD

Flags: not altered

STOSB

Logic:

```
(ES:DI) 🡐 AL
if DF = 0
        DI 🡐 DI + 1
else
        DI 🡐 DI - 1
```

STOSB copies the value in AL into the location pointed to by ES:DI. DI is then incremented (i f the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing AL in the next location.

Operands	Clocks	Transfers	Bytes	Example
-	11	1	1	STOSB

(repeat)	9+10/rep	1/rep	1	REP STOSB

When used in conjunction wit h the REP prefixes, the Store String instructions are useful for initializing a block of memory. For example, the following code would initialize the 100-byte memory block at BUFFER to 0:

```
MOV AL,0              ;The value to initialize BUFFER to
LEA DI,BUFFER         ;Starting location of BUFFER
MOV CX,100            ;Size of BUFFER
CLD                   ;Let's move in forward direction
REP STOSB             ;Compare this line to example for STOS
```

STOSW Store String Word

See Also: STOS, STOSB, CMPS, LODS, MOVS, SCAS, REP, CLD, STD

Flags: not altered

Logic:

```
(ES:DI) ← AX
if DF = 0
        DI ← DI + 2
else
        DI ← DI - 2
```

STOSW copies the value in AX into the location pointed to by ES:DI. DI is then incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing AX in the next location.

Operands	Clocks	Transfers	Bytes	Example
-	15	1	1	STOSW
(repeat)	9+14/rep	1/rep	1	REP STOSW

When used in conjunction with the REP prefixes, the Store String instructions are useful for initializing a block of memory. For example, the following code would initialize the 100-byte memory block at BUFFER to 0:

```
MOV AX,0              ;The value to initialize BUFFER to
LEA DI,BUFFER         ;Starting location of BUFFER
MOV CX,50             ;Size of BUFFER, in words
CLD                   ;Let's move in forward direction
REP STOSW             ;Compare this line to example for STOS
```

SUB Subtract

See Also: SBB, DEC, NEG, CMP, AAS, DAS, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

SUB destination,source

Logic: destination ← destination - source

SUB subtracts the source operan d from the destination operand and stores the result in destination. Both operands may be bytes or words, and both may signed or unsigned binary numbers.

Operands	Clocks byte(word)	Transfers	Bytes	Example
register, register	3	-	2	SUB DX,BX
register, memory	9(13)+EA	1	2-4	SUB DX,TOTAL
memory, register	16(24)+EA	2	2-4	SUB RATE,CL
accumulator, immediate	4	-	2-3	SUB AH,25
register, immediate	4	-	3-4	SUB DX,5280
memory, immediate	17(25)+EA	2	3-6	SUB RESULT,1032

Note: You may wish to use SBB if you need to subtract numbers that are larger than 16 bits, since SBB subtracts a borrow from a previous operation.

You may subtract a byte -length immediate value from a destination which is a word; in this case, the byte is sign - extended to 16 bits before the subtraction.

TEST Test

See Also: CMP, AND, NOT, OR, XOR, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

TEST destination,source

Logic:

```
(destination AND source)      ; Set flags only
CF ← 0
OF ← 0
```

TEST performs a logical AND on its two operands and updates the flags. Neither the destination nor source is changed.

Operands	Clocks	Transfers	Bytes	Example

	byte(word)			
register, register	3	-	2	TEST SI,DX
register, memory	9(13)+EA	1	2-4	TEST SI,MASK
accumulator, immediate	4	-	2-3	TEST AL,00000100b
register, immediate	5	-	3-4	TEST CX,1234
memory, immediate	11+EA	-	3-6	TEST PARAM,1F1Fh

TEST is useful for examining the status of individual bits. For example, the following section of co de will transfer control to ONE_FIVE_OFF if both bits one and five of register AL are cleared. The status of all other bits will be ignored.

```
          TEST AL,00100010b               ;Mask out all bits except 1 and 5
          JZ      ONE_FIVE_OFF            ;If either was set, result was not 0
    NOT_BOTH:                  .          ;One or both bits was set
    ONE_FIVE_OFF:                         ;Bits 1 and 5 were off
                               .
                               .
```

WAIT Wait

See Also: HLT, ESC, LOCK

Flags: not altered

Logic: None

WAIT causes the processor to enter a wait state. The processor will remain inactive until the TEST input on the microprocessor is driven active.

Operands	Clocks	Transfers	Bytes	Example
no operands	3+5n	-	1	WAIT

Note: This instruction is used to synchronize external hardware, such as a coprocessor.

XCHG Exchange Registers

See Also: MOV, PUSH, POP, XLAT, EA

Flags: not altered

XCHG destination,source

Logic: destination <-> source

XCHG switches the contents of its operands, which may be either bytes or words.

Operands	Clocks	Transfers	Bytes	Example
	byte(word)			
accumulator, reg16	3	-	1	XCHG AX,SI
memory, register	17(25)+EA	2	2-4	LOCK XCHG SEMPHOR, DX
register, register	4	-	2	XCHG CL,DL

Note: Used in conjunction with the LOCK prefix, this instruction is particularly useful when implementing semaphores to control shared resources.

XLAT Translate

Flags: not altered

XLAT translate-table

Logic: AL $\leftarrow$ (BX + AL)

XLAT translates bytes via a table lookup. A pointer to a 256-byte translation table is loaded into BX. The byte to be translated is loaded into AL; it serves as an index (ie, offset) into the translation table. After the XLAT instruction is ex ecuted, the byte in AL is replaced by the byte located AL bytes from the beginning of the translate-table.

Operands	Clocks	Transfers	Bytes	Example
translate-table	11	1	1	XLAT SINE_TABLE

Note: Translate-table can be less than 256 bytes.

The operand, t ranslate-table, is optional since a pointer to the table must be loaded into BX before the instruction is executed.

The following example translates a decimal value (0 to 15) to the corresponding single hexadecimal digit.

```
          LEA BX, HEX_TABLE               ;pointer to table into BX
          MOV AL, DECIMAL_DIGIT           ;digit to be translated to AL
          XLAT HEX_TABLE                  ;translate the value in AL
                                          ;AL now contains ASCII hex digit
                               .
          HEX_TABLE DB '0123456789ABCDEF'
```

XOR Exclusive OR

See also:OR, AND, NOT, EA, Flags

Flags Affected:

O	D	I	T	S	Z	A	P	C
0				*	*	*	*	0

XOR destination,source

Logic: destination ← destination XOR source

XOR performs a bit -by-bit "exclusive or" on its two operands, and returns the result in the destination operand.The operands may be bytes or words.

XOR Instruction Logic

Destination	Source	Result
0	0	0
0	1	1
1	0	1
1	1	0

XOR sets each bit of the result to 1 only one of the corresponding bits is set to one.

Operands	Clocks	Transfers	Bytes	Example
register, register	3	-	2	XOR CX,BX
register, memory	9(13)+EA	1	2-4	XOR CL,MASK_BYTE
memory, register	16(24)+EA	2	2-4	XOR ALPHA[SI],DX
accumulator, immediate	4	-	2-3	XOR AL,01000001b
register, immediate	4	-	3-4	XOR SI,00C2h
memory, immediate	17(25)+EA	2	3-6	XOR RETURN_CODE,0D2h

FIM