C H A P T E R 4

Processor

Topical Cross-reference for Processor Instructions

Arithmetic

ADC	ADD	DEC
DIV	IDIV	IMUL
INC	MUL	NEG
SBB	SUB	XADD#

BCD Conversion

AAA	AAD	AAM
AAS	DAA	DAS

Bit Operations

AND	BSF§	BSR§
BT§	BTC§	BTR§
BTS§	NOT	OR
RCL	RCR	ROL
ROR	SAR	SHL/SAL
SHLD§	SHR	SHRD§
XOR		

Compare

BT§	BTC§	BTR§
BTS§	CMP	CMPS
CMPXCHG#	TEST	

Conditional Set

SETA/SETNBE§	SETAE/SETNB§	SETB/SETNAE§
SETBE/SETNA§	SETC§	SETE/SETZ§
SETG/SETNLE§	SETGE/SETNL§	SETL/SETNGE§
SETLE/SETNG§	SETNC§	SETNE/SETNZ§
SETNO§	SETNP/SETPO§	SETNS§
SETO§	SETP/SETPE§	SETS§

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Conditional Transfer

BOUND*	INTO	JA/JNBE
JAE/JNB	JB/JNAE	JBE/JNA
JC	JCXZ/JECXZ	JE/JZ
JG/JNLE	JGE/JNL	JL/JNGE
JLE/JNG	JNC	JNE/JNZ
JNO	JNP/JPO	JNS
JO	JP/JPE	JS

Data Transfer

BSWAP#	CMPXCHG#	LDS/LES
LEA	LFS/LGS/LSS§	LODS
MOV	MOVS	MOVSX§
MOVZX§	STOS	XADD#
XCHG	XLAT/XLATB	

Flag

CLC	CLD	CLI
CMC	LAHF	POPF
PUSHF	SAHF	STC
STD	STI	

Input/Output

IN	INS*
OUT	OUTS*

Loop

JCXZ/JECXZ	LOOP
LOOPE/LOOPZ	LOOPNE/LOOPNZ

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Process Control

ARPL†	CLTS†	LAR†
LGDT/LIDT/LLDT†	LMSW†	LSL†
LTR†	SGDT/SIDT/SLDT†	SMSW†
STR†	VERR†	VERW†
MOV *special*§	INVD#	INVLPG#
WBINVD#		

Processor Control

HLT	LOCK
NOP	WAIT

Stack

PUSH	PUSHF	PUSHA*
PUSHAD*	POP	POPF
POPA*	POPAD*	ENTER*
LEAVE*		

String

MOVS	LODS	STOS
SCAS	CMPS	INS*
OUTS*	REP	REPE/REPZ
REPNE/REPNZ		

Type Conversion

CBW	CWD
CWDE§	CDQ§
BSWAP#	

Unconditional Transfer

CALL	INT	IRET
RET	RETN/RETF	JMP

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Interpreting Processor Instructions

The following sections explain the format of instructions for the 8086, 8088, 80286, 80386, and 80486 processors. Those instructions begin on page 64.

Flags

Only the flags common to all processors are shown. If none of the flags is affected by the instruction, the flag line says No change. If flags can be affected, a two-line entry is shown. The first line shows flag abbreviations as follows:

Abbreviation	**Flag**
O	Overflow
D	Direction
I	Interrupt
T	Trap
S	Sign
Z	Zero
A	Auxiliary carry
P	Parity
C	Carry

The second line has codes indicating how the flag can be affected:

Code	**Effect**
1	Sets the flag
0	Clears the flag
?	May change the flag, but the value is not predictable
blank	No effect on the flag
±	Modifies according to the rules associated with the flag

Syntax

Each encoding variation may have different syntaxes corresponding to different addressing modes. The following abbreviations are used:

reg A general-purpose register of any size.

segreg One of the segment registers: DS, ES, SS, or CS (also FS or GS on the 80386–80486).

accum An accumulator register of any size: AL or AX (also EAX on the 80386–80486).

mem A direct or indirect memory operand of any size.

label A labeled memory location in the code segment.

src,dest A source or destination memory operand used in a string operation.

immed A constant operand.

In some cases abbreviations have numeric suffixes to specify that the operand must be a particular size. For example, *reg16* means that only a 16-bit (word) register is accepted.

Examples

One or more examples are shown for each syntax. Their position is not related to the clock speeds in the right column.

Clock Speeds

Column 3 shows the clock speeds for each processor. Sometimes an instruction may have more than one clock speed. Multiple speeds are separated by commas. If several speeds are part of an expression, they are enclosed in parentheses. The following abbreviations are used to specify variations:

EA Effective address. This applies only to the 8088 and 8086 processors, as described in the next section.

b,w,d Byte, word, or doubleword operands.

pm Protected mode.

n Iterations. Repeated instructions may have a base number of clocks plus a number of clocks for each iteration. For example, 8+4*n* means 8 clocks plus 4 clocks for each iteration.

noj No jump. For conditional jump instructions, *noj* indicates the speed if the condition is false and the jump is not taken.

m Next instruction components. Some control transfer instructions take different times depending on the length of the next instruction executed. On the 8088 and 8086, *m* is never a factor. On the 80286, *m* is the number of bytes in the instruction. On the 80386–80486, *m* is the number of components. Each byte of encoding is a component, and the displacement and data are separate components.

W88,88 8088 exceptions. See “Timings on the 8088 and 8086 Processors,” following.

Clocks can be converted to nanoseconds by dividing 1 microsecond by the number of megahertz (MHz) at which the processor is running. For example, on a processor running at 8 MHz, 1 clock takes 125 nanoseconds (1000 MHz per nanosecond / 8 MHz).

The clock counts are for best-case timings. Actual timings vary depending on wait states, alignment of the instruction, the status of the prefetch queue, and other factors.

Timings on the 8088 and 8086 Processors

Because of its 8-bit data bus, the 8088 always requires two fetches to get a 16-bit operand. Therefore, instructions that work on 16-bit memory operands take longer on the 8088 than on the 8086. Separate 8088 timings are shown in parentheses following the main timing. For example, 9 (*W88*=13) means that the 8086 with any operands or the 8088 with byte operands take 9 clocks, but the 8088 with word operands takes 13 clocks. Similarly, 16 (*88*=24) means that the 8086 takes 16 clocks, but the 8088 takes 24 clocks.

On the 8088 and 8086, the effective address (*EA*) value must be added for instructions that operate on memory operands. A displacement is any direct memory or constant operand, or any combination of the two. The following shows the number of clocks to add for the effective address:

Components	EA Clocks	Examples

Displacement	6	`mov ax,stuff`
		`mov ax,stuff+2`

Base or index	5	`mov ax,[bx]`
		`mov ax,[di]`

Displacement plus base or index	9	`mov ax,[bp+8]`
		`mov ax,stuff[di]`

Base plus index (BP+DI, BX+SI)	7	`mov ax,[bx+si]`
		`mov ax,[bp+di]`

Components	EA Clocks	Examples

Base plus index (BP+SI, BX+DI)	8	`mov ax,[bx+di]`
		`mov ax,[bp+si]`

Base plus index plus displacement (BP+DI+*disp*, BX+SI+*disp*)	11	`mov ax,stuff[bx+si]`
		`mov ax,[bp+di+8]`

Base plus index plus displacement (BP+SI+*disp*, BX+DI+*disp*)	12	`mov ax,stuff[bx+di]`
		`mov ax,[bp+si+20]`

Segment override	*EA*+2	`mov ax,es:stuff`
		`mov ax,ds:[bp+10]`

Timings on the 80286–80486 Processors

On the 80286–80486 processors, the effective address calculation is handled by hardware and is therefore not a factor in clock calculations except in one case. If a memory operand includes all three possible elements—a displacement, a base register, and an index register—then add one clock. On the 80486, the extra clock is not always used. Examples are shown in the following.

```
mov     ax,[bx+di]                ;No extra

mov     ax,array[bx+di]           ;One extra

mov     ax,[bx+di+6]              ;One extra
```

Note 80186 and 80188 timings are different from 8088, 8086, and 80286 timings. They are not shown in this manual. Timings are also not shown for protected-mode transfers through gates or for the virtual 8086 mode available on the 80386–80486 processors.

Interpreting Encodings

Encodings are shown for each variation of the instruction. This section describes encoding for all processors except the 80386–80486. The encodings take the form of boxes filled with 0s and 1s for bits that are constant for the instruction variation, and abbreviations (in italics) for the following variable bits or bitfields:

d Direction bit. If set, do memory to register; the *reg* field is the destination. If clear, do register to memory or register to register; the *reg* field is the source.

a Accumulator direction bit. If set, move accumulator register to memory. If clear, move memory to accumulator register.

w Word/byte bit. If set, use 16-bit or 32-bit operands. If clear, use 8-bit operands.

s Sign bit. If set, sign-extend 8-bit immediate data to 16 bits.

mod Mode. This 2-bit field gives the register/memory mode with displacement. The possible values are shown below:

mod	**Meaning**
00	This value can have two meanings: If r/m is 110, a direct memory operand is used. If r/m is not 110, the displacement is 0 and an indirect memory operand is used. The operand must be based, indexed, or based indexed.
01	An indirect memory operand is used with an 8-bit displacement.
10	An indirect memory operand is used with a 16-bit displacement.
11	A two-register instruction is used; the *reg* field specifies the destination and the *r/m* field specifies the source.

reg Register. This 3-bit field specifies one of the general-purpose registers:

reg	**16/32-bit if *w*=1**	**8-bit if *w*=0**
000	AX/*EA*X	AL
001	CX/ECX	CL
010	DX/EDX	DL
011	BX/EBX	BL
100	SP/ESP	AH
101	BP/EBP	CH
110	SI/ESI	DH
111	DI/EDI	BH

The *reg* field is sometimes used to specify encoding information rather than a register.

sreg Segment register. This field specifies one of the segment registers:

sreg	**Register**
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

r/m Register/memory. This 3-bit field specifies a register or memory *r/m* operand.

If the *mod* field is 11, *r/m* specifies the source register using the *reg* field codes. Otherwise, the field has one of the following values:

r/m	**Operand Address**
000	DS:[BX+SI+*disp*]
001	DS:[BX+DI+*disp*]
010	SS:[BP+SI+*disp*]
011	SS:[BP+DI+*disp*]
100	DS:[SI+*disp*]
101	DS:[DI+*disp*]
110	SS:[BP+*disp*]*
111	DS:[BX+*disp*]

* If *mod* is 00 and *r/m* is 110, then the operand is treated as a direct memory operand. This means that the operand `[BP]` is encoded as `[BP+0]` rather than having a short-form like other register indirect operands. Encoding `[BX]` takes one byte, but encoding `[BP]` takes two.

disp Displacement. These bytes give the offset for memory operands. The possible lengths (in bytes) are shown in parentheses.

data Data. These bytes give the actual value for constant values. The possible lengths (in bytes) are shown in parentheses.

If a memory operand has a segment override, the entire instruction has one of the following bytes as a prefix:

Prefix	**Segment**
00101110 (2Eh)	CS
00111110 (3Eh)	DS
00100110 (26h)	ES
00110110 (36h)	SS
01100100 (64h)	FS
01100101 (65h)	GS

Example

As an example, assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable):

```
        add     warray[bx+di], -3
```

First look up the encoding for the immediate-to-memory syntax of the **ADD** instruction:

100000*sw* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (0, 1, or 2)*

Since the destination is a word operand, the *w* bit is set. The 8-bit immediate data must be sign-extended to 16 bits to fit into the operand, so the *s* bit is also set. The first byte of the instruction is therefore 10000011 (83h).

Since the memory operand can be anywhere in the segment, it must have a 16-bit offset (displacement). Therefore the *mod* field is 10. The *reg* field is 000, as shown in the encoding. The *r/m* coding for [bx+di+*disp*] is 001. The second byte is 10000001 (81h).

The next two bytes are the offset of `warray`. The low byte of the offset is stored first and the high byte second. For this example, assume that `warray` is located at offset 10EFh.

The last byte of the instruction is used to store the 8-bit immediate value –3 (FDh). This value is encoded as 8 bits (but sign-extended to 16 bits by the processor).

The encoding is shown here in hexadecimal:

83 81 EF 10 FD

You can confirm this by assembling the instruction and looking at the resulting assembly listing.

Interpreting 80386–80486 Encoding Extensions

This book shows 80386–80486 encodings for instructions that are available only on the 80386–80486 processors. For other instructions, encodings are shown only for the 16-bit subset available on all processors. This section tells how to convert the 80286 encodings shown in the book to 80386–80486 encodings that use extensions such as 32-bit registers and memory operands.

The extended 80386–80486 encodings differ in that they can have additional prefix bytes, a Scaled Index Base (SIB) byte, and 32-bit displacement and immediate bytes. Use of these elements is closely tied to the segment word size. The use type of the code segment determines whether the instructions are processed in 32-bit mode (**USE32**) or 16-bit mode (**USE16**). Current versions of MS-DOS® and Microsoft® Windows™ use 16-bit mode only. Windows NT uses 32-bit mode.

The bytes that can appear in an instruction encoding are:

16-Bit Encoding

Opcode	***mod-reg-r/m***	***disp***	***immed***
(1-2)	(0-1)	(0-2)	(0-2)

32-Bit Encoding

Address-Size (67h)	**Operand-Size (66h)**	**Opcode**	***mod-reg-r/m***	**Scaled Index Base**	***disp***	***immed***
(0-1)	(0-1)	(1-2)	(0-1)	(0-1)	(0-4)	(0-4)

Additional bytes may be added for a segment prefix, a repeat prefix, or the **LOCK** prefix.

Address-Size Prefix

The address-size prefix determines the segment word size of the operation. It can override the default size for calculating the displacement of memory addresses. The address prefix byte is 67h. The assembler automatically inserts this byte where appropriate.

In 32-bit mode (**USE32** or **FLAT** code segment), displacements are calculated as 32-bit addresses. The effective address-size prefix must be used for any instructions that must calculate addresses as 16-bit displacements. In 16-bit mode, the defaults are reversed. The prefix must be used to specify calculation of 32-bit displacements.

Operand-Size Prefix

The operand-size prefix determines the size of operands. It can override the default size of registers or memory operands. The operand-size prefix byte is 66h. The assembler automatically inserts this byte where appropriate.

In 32-bit mode, the default sizes for operands are 8 bits and 32 bits (depending on the *w* bit). For most instructions, the operand-size prefix must be used for any instructions that use 16-bit operands. In 16-bit mode, the default sizes are 8 bits and 16 bits. The prefix must be used for any instructions that use 32-bit operands. Some instructions use 16-bit operands, regardless of mode.

Encoding Differences for 32-Bit Operations

When 32-bit operations are performed, the meaning of certain bits or fields is different from their meaning in 16-bit operations. The changes may affect default operations in 32-bit mode, or 16-bit mode operations in which the address-size prefix or the operand-size prefix is used. The following fields may have a different meaning for 32-bit operations from their meaning as described in the “Interpreting Encodings” section:

w Word/byte bit. If set, use 32-bit operands. If clear, use 8-bit operands.

s Sign bit. If set, sign-extend 8-bit and 16-bit immediate data to 32 bits.

mod Mode. This field indicates the register/memory mode. The value 11 still indicates a register-to-register operation with *r/m* containing the code for a 32-bit source register. However, other codes have different meanings as shown in the tables in the next section.

reg Register. The codes for 16-bit registers are extended to 32-bit registers. For example, if the *reg* field is 000, EAX is used instead of AX. Use of 8-bit registers is unchanged.

sreg Segment register. The 80386 has the following additional segment registers:

sreg	**Register**
100	FS
101	GS

r/m Register/memory. If the *r/m* field is used for the source register, 32-bit registers are used as for the *reg* field. If the field is used for memory operands, the meaning is completely different from the meaning used for 16-bit operations, as shown in the tables in the next section.

disp Displacement. This field is 4 bytes for 32-bit addresses.

data Data. Immediate data can be up to 4 bytes.

Scaled Index Base Byte

Many 80386–80486 extended memory operands are too complex to be represented by a single *mod-reg-r/m* byte. For these operands, a value of 100 in the *r/m* field signals the presence of a second encoding byte called the Scaled Index Base (SIB) byte. The SIB byte is made up of the following fields:

ss index base

ss Scaling Field. This two-bit field specifies one of the following scaling factors:

ss	**Scale**
00	1
01	2
10	4

11	8

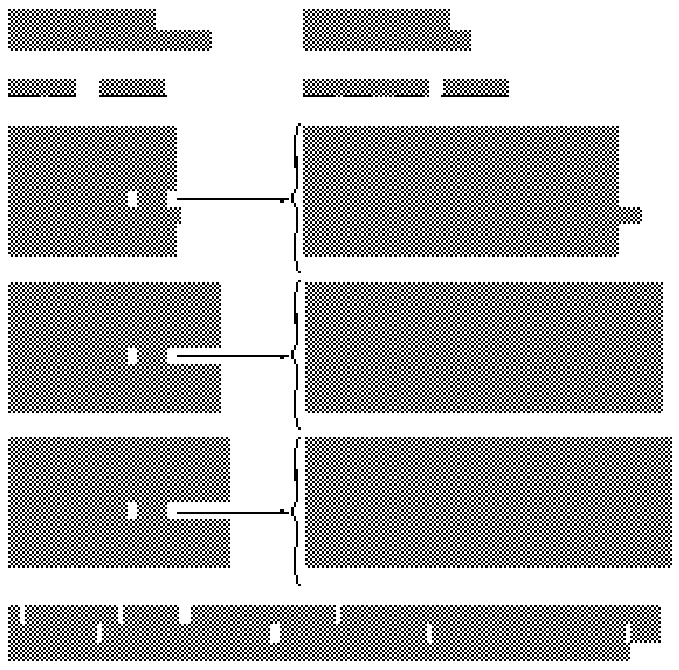
index Index Register. This three-bit field specifies one of the following index registers:

index	**Register**
000	*EA*X
001	ECX
010	EDX
011	EBX
100	no index
101	EBP
110	ESI
111	EDI

Note ESP cannot be an index register. If the *index* field is 100, the *ss* field must be 00.

base Base Register. This 3-bit field combines with the *mod* field to specify the base register and the displacement. Note that the *base* field only specifies the base when the *r/m* field is 100. Otherwise, the *r/m* field specifies the base.

The possible combinations of the mod, r/m, scale, index, and base fields are as follows:



If a memory operand has a segment override, the entire instruction has one of the prefixes discussed in the preceding section, “Interpreting Encodings,” or one of the following prefixes for the segment registers available only on the 80386–80486:

Prefix	Segment
01100100 (64h)	FS
01100101 (65h)	GS

Example

Assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable). Assume that the instruction is used in 16-bit mode.

```
        add     warray[eax+ecx*2], -3
```

First look up the encoding for the immediate-to-memory syntax of the **ADD** instruction:

100000*sw* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

This encoding must be expanded to account for 80386–80486 extensions. Note that the instruction operates on 16-bit data in a 16-bit mode program. Therefore, the operand-size prefix is not needed. However, the instruction does use 32-bit registers to calculate a 32-bit effective address. Thus the first byte of the encoding must be the effective address-size prefix, 01100111 (67h).

The *opcode* byte is the same (83h) as for the 80286 example described in the "Interpreting Encodings" section.

The *mod-reg-r/m* byte must specify a based indexed operand with a scaling factor of two. This operand cannot be specified with a single byte, so the encoding must also use the SIB byte. The value 100 in the *r/m* field specifies an SIB byte. The *reg* field is 000, as shown in the encoding. The *mod* field is 10 for operands that have base and scaled index registers and a 32-bit displacement. The combined *mod*, *reg*, and *r/m* fields for the second byte are 10000100 (84h).

The SIB byte is next. The scaling factor is 2, so the *ss* field is 01. The index register is ECX, so the *index* field is 001. The base register is EAX, so the *base* field is 000. The SIB byte is 01001000 (48h).

The next 4 bytes are the offset of `warray`. The low bytes are stored first. For this example, assume that `warray` is located at offset 10EFh. This offset only requires 2 bytes, but 4 must be supplied because of the addressing mode. A 32-bit address can be safely used in 16-bit mode as long as the upper word is 0.

The last byte of the instruction is used to store the 8-bit immediate value –3 (FDh). The encoding is shown here in hexadecimal:

67 83 84 48 00 00 EF 10 FD

Instructions

This section provides an alphabetical reference to the instructions for the 8086, 8088, 80286, 80386, and 80486 processors.

AAA ASCII Adjust After Addition

Adjusts the result of an addition to a decimal digit (0–9). The previous addition instruction should place its 8-bit sum in AL. If the sum is greater than 9h, AH is incremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Encoding 00110111

Syntax	Examples	CPU	Clock Cycles
AAA	`aaa`	88/86	8
		286	3
		386	4
		486	3

AAD ASCII Adjust Before Division

Converts unpacked BCD digits in AH (most significant digit) and AL (least significant digit) to a binary number in AX. This instruction is often used to prepare an unpacked BCD number in AX for division by an unpacked BCD digit in an 8-bit register.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Encoding 11010101 00001010

Syntax	Examples	CPU	Clock Cycles
AAD	`aad`	88/86	60
		286	14
		386	19
		486	14

AAM ASCII Adjust After Multiply

Converts an 8-bit binary number less than 100 decimal in AL to an unpacked BCD number in AX. The most significant digit goes in AH and the least significant in AL. This instruction is often used to adjust the product after a **MUL** instruction that multiplies unpacked BCD digits in AH and AL. It is also used to adjust the quotient after a **DIV** instruction that divides a binary number less than 100 decimal in AX by an unpacked BCD number.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Encoding 11010100 00001010

Syntax	Examples	CPU	Clock Cycles
AAM	`aam`	88/86	83
		286	16
		386	17
		486	15

AAS ASCII Adjust After Subtraction

Adjusts the result of a subtraction to a decimal digit (0–9). The previous subtraction instruction should place its 8-bit result in AL. If the result is greater than 9h, AH is decremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Encoding 00111111

Syntax	Examples	CPU	Clock Cycles
AAS	`aas`	88/86	8
		286	3
		386	4
		486	3

ADC Add with Carry

Adds the source operand, the destination operand, and the value of the carry flag. The result is assigned to the destination operand. This instruction is used to add the more significant portions of numbers that must be added in multiple registers.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

000100*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *reg,reg*	`adc dx,cx`	88/86	3
		286	2
		386	2
		486	1
ADC *mem,reg*	`adc WORD PTR m32[2],dx`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	7
		486	3
ADC *reg,mem*	`adc dx,WORD PTR m32[2]`	88/86	9+*EA* (*W88*=13+*EA*)
		286	7
		386	6
		486	2

Encoding

100000*sw* *mod, 010,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *reg,immed*	`adc dx,12`	88/86	4
		286	3
		386	2
		486	1
ADC *mem,immed*	`adc WORD PTR m32[2],16`	88/86	17+*EA* (*W88*=23+*EA*)
		286	7
		386	7
		486	3

Encoding

0001010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *accum,immed*	`adc ax,5`	88/86	4
		286	3
		386	2
		486	1

ADD Add

Adds the source and destination operands and puts the sum in the destination operand.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 000000*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ADD *reg,reg*	`add ax,bx`	88/86	3
		286	2
		386	2
		486	1
ADD *mem, reg*	`add total, cx`	88/86	16+*EA* (*W88*=24+*EA*)
	`add array[bx+di], dx`	286	7
		386	7
		486	3
ADD *reg,mem*	`add cx,incr`	88/86	9+*EA* (*W88*=13+*EA*)
	`add dx,[bp+6]`	286	7
		386	6
		486	2

Encoding 100000*sw* *mod, 000,r/m* *disp (p,1, or2)* *data (1or2)*

Syntax	Examples	CPU	Clock Cycles
ADD *reg,immed*	`add bx,6`	88/86	4
		286	3
		386	2
		486	1
ADD *mem,immed*	`add amount,27`	88/86	17+*EA* (*W88*=23+*EA*)
	`add pointers[bx][si],6`	286	7
		386	7
		486	3

Encoding 0000010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADD *accum,immed*	`add ax,10`	88/86	4
		286	3
		386	2
		486	1

AND Logical AND

Performs a bitwise AND operation on the source and destination operands and stores the result in the destination operand. For each bit position in the operands, if both bits are set, the corresponding bit of the result is set. Otherwise, the corresponding bit of the result is cleared.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding

001000*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *reg,reg*	`and dx,bx`	88/86	3
		286	2
		386	2
		486	1
AND *mem,reg*	`and bitmask,bx`	88/86	16+*EA* (*W88*=24+*EA*)
	`and [bp+2],dx`	286	7
		386	7
		486	3
AND *reg,mem*	`and bx,masker`	88/86	9+*EA* (*W88*=13+*EA*)
	`and dx,marray[bx+di]`	286	7
		386	6
		486	2

Encoding

100000*sw* *mod, 100, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *reg,immed*	`and dx,0F7h`	88/86	4
		286	3
		386	2
		486	1
AND *mem,immed*	`and masker, 100lb`	88/86	17+*EA*(*W88*=24+*EA*)
		286	7
		386	7
		486	3

Encoding

0010010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *accum,immed*	`and ax,0B6h`	88/86	4
		286	3
		386	2
		486	1

ARPL Adjust Requested Privilege Level

80286–80486 Protected Only Verifies that the destination Requested Privilege Level (RPL) field (bits 0 and 1 of a selector value) is less than the source RPL field. If it is not, **ARPL** adjusts the destination RPL to match the source RPL. The destination operand should be a 16-bit memory or register operand containing the value of a selector. The source operand should be a 16-bit register containing the test value. The zero flag is set if the destination is adjusted; otherwise, the flag is cleared. **ARPL** is useful only in 80286–80486 protected mode. See Intel documentation for details on selectors and privilege levels.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding

01100011 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ARPL *reg,reg*	`arpl ax,cx`	88/86	—
		286	10
		386	20
		486	9
ARPL *mem,reg*	`arpl selector,dx`	88/86	—
		286	11
		386	21
		486	9

BOUND Check Array Bounds

80286–80486 Only Verifies that a signed index value is within the bounds of an array. The destination operand can be any 16-bit register containing the index to be checked. The source operand must then be a 32-bit memory operand in which the low and high words contain the starting and ending values, respectively, of the array. (On the 80386–80486 processors, the destination operand can be a 32-bit register; in this case, the source operand must be a 64-bit operand made up of 32-bit bounds.) If the source operand is less than the first bound or greater than the last bound, an interrupt 5 is generated. The instruction pointer pushed by the interrupt (and returned by **IRET**) points to the **BOUND** instruction rather than to the next instruction.

Flags

No change

Encoding 01100010 *mod,reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
BOUND *reg16,mem32*	`bound di,base-4`	88/86	—
BOUND *reg32,mem64**		286	*noj*=13†
		386	*noj*=10†
		486	*noj*=7

* 80386–80486 only.

† See **INT** for timings if interrupt 5 is called.

BSF/BSR Bit Scan

80386–80486 Only Scans an operand to find the first set bit. If a set bit is found, the zero flag is cleared and the destination operand is loaded with the bit index of the first set bit encountered. If no set bit is found, the zero flag is set. **BSF** (Bit Scan Forward) scans from bit 0 to the most significant bit. **BSR** (Bit Scan Reverse) scans from the most significant bit of an operand to bit 0.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 10111100 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BSF *reg16,reg16*	`bsf cx,bx`	88/86	—
BSF *reg32,reg32*		286	—
		386	10+3*n**
		486	6–42†
BSF *reg16,mem16*	`bsf ecx,bitmask`	88/86	—
BSF *reg32,mem32*		286	—
		386	10+3*n**
		486	7–43§

Encoding 00001111 10111101 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BSR *reg16,reg16*	`bsr cx,dx`	88/86	—
BSR *reg32,reg32*		286	—
		386	10+3*n**
		486	103 – 3*n*#
BSR *reg16,mem16*	`bsr eax,bitmask`	88/86	—
BSR *reg32,mem32*		286	—
		386	10+3*n**
		486	104 – 3*n*#

* *n* = bit position from 0 to 31.
clocks = 6 if second operand equals 0.

† Clocks = 8 +
4 for each byte scanned +
3 for each nibble scanned +
3 for each bit scanned in last nibble
or 6 if second operand equals 0.

§ Same as footnote above, but add 1 clock.

# *n* = bit position from 0 to 31.
clocks = 7 if second operand equals 0.

BSWAP Byte Swap

80486 Only Takes a single 32-bit register as operand and exchanges the first byte with the fourth, and the second byte with the third. This instruction does not alter any bit values within the bytes and is useful for quickly translating between 8086-family byte storage and storage schemes in which the high byte is stored first.

Flags No change

Encoding 00001111 11001 *reg*

Syntax	Examples	CPU	Clock Cycles
BSWAP *reg32*	`bswap eax`	88/86	—
	`bswap ebx`	286	—
		386	—
		486	1

BT/BTC/BTR/BTS Bit Tests

80386–80486 Only Copies the value of a specified bit into the carry flag, where it can be tested by a **JC** or **JNC** instruction. The destination operand specifies the value in which the bit is located; the source operand specifies the bit position. **BT** simply copies the bit to the flag. **BTC** copies the bit and complements (toggles) it in the destination. **BTR** copies the bit and resets (clears) it in the destination. **BTS** copies the bit and sets it in the destination.

Flags

O	D	I	T	S	Z	A	P	C
								±

Encoding

00001111 10111010 *mod, BBB*,r/m* *disp (0, 1, 2, or 4)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
BT *reg16,immed8†*	`bt ax,4`	88/86	—
		286	—
		386	3
		486	3
BTC *reg16,immed8†*	`bts ax,4`	88/86	—
BTR *reg16,immed8†*	`btr bx,17`	286	—
BTS *reg16,immed8†*	`btc edi,4`	386	6
		486	6
BT *mem16,immed8†*	`btr DWORD PTR`	88/86	—
	`[si],27`	286	—
	`btc color[di],4`	386	6
		486	3
BTC *mem16,immed8†*	`btc DWORD PTR`	88/86	—
BTR *mem16,immed8†*	`[bx],27`	286	—
BTS *mem16,immed8†*	`btc maskit,4`	386	8
	`btr color[di],4`	486	8

Encoding

00001111 10*BBB*011* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BT *reg16,reg16†*	`bt ax,bx`	88/86	—
		286	—
		386	3
		486	3
BTC *reg16,reg16†*	`btc eax,ebx`	88/86	—
BTR *reg16,reg16†*	`bts bx,ax`	286	—
BTS *reg16,reg16†*	`btr cx,di`	386	6
		486	6

Syntax	Examples	CPU	Clock Cycles
BT *mem16,reg16*†	`bt [bx],dx`	88/86	—
		286	—
		386	12
		486	8
BTC *mem16,reg16*†	`bts flags[bx],cx`	88/86	—
BTR *mem16,reg16*†	`btr rotate,cx`	286	—
BTS *mem16,reg16*†	`btc [bp+8],si`	386	13
		486	13

* *BBB* is 100 for **BT**, 111 for **BTC**, 110 for **BTR**, and 101 for **BTS**.

† Operands also can be 32 bits (*reg32* and *mem32*).

CALL Call Procedure

Calls a procedure. The instruction pushes the address of the next instruction onto the stack and jumps to the address specified by the operand. For **NEAR** calls, the offset (IP) is pushed and the new offset is loaded into IP.

For **FAR** calls, the segment (CS) is pushed and the new segment is loaded into CS. Then the offset (IP) is pushed and the new offset is loaded into IP. A subsequent **RET** instruction can pop the address so that execution continues with the instruction following the call.

Flags No change

Encoding 11101000 *disp (2)*

Syntax	Examples	CPU	Clock Cycles
CALL *label*	`call upcase`	88/86	19 (*88*=23)
		286	7+*m*
		386	7+*m*
		486	3

Encoding 10011010 *disp (4)*

Syntax	Examples	CPU	Clock Cycles
CALL *label*	`call FAR PTR job`	88/86	28 (*88*=36)
	`call distant`	286	13+*m*,*pm*=26+*m**
		386	17+*m*,*pm*=34+*m**
		486	18,*pm*=20*

Encoding 11111111 *mod,010,r/m*

Syntax	Examples	CPU	Clock Cycles
CALL *reg*	`call ax`	88/86	16 (*88*=20)
		286	7+*m*
		386	7+*m*
		486	5
CALL *mem16*	`call pointer`	88/86	21+*EA* (*88*=29+*EA*)
CALL *mem32*†	`call [bx]`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11111111 *mod*,011,*r/m*

Syntax	Examples	CPU	Clock Cycles
CALL *mem32*	`call far_table[di]`	88/86	37+*EA* (*88*=53+*EA*)
CALL *mem48*†	`call DWORD PTR [bx]`	286	16+*m*,*pm*=29+*m**
		386	22+*m*,*pm*=38+*m**
		486	17,*pm*=20*

* Timings for calls through call and task gates are not shown, since they are used primarily in operating systems.

† 80386–80486 32-bit addressing mode only.

CBW Convert Byte to Word

Converts a signed byte in AL to a signed word in AX by extending the sign bit of AL into all bits of AH.

Flags No change

Encoding 10011000*

Syntax	Examples	CPU	Clock Cycles
CBW	`cbw`	88/86	2
		286	2
		386	3
		486	3

* **CBW** and **CWDE** have the same encoding with two exceptions: in 32-bit mode, **CBW** is preceded by the operand-size byte (66h) but **CWDE** is not; in 16-bit mode, **CWDE** is preceded by the operand-size byte but **CBW** is not.

CDQ Convert Double to Quad

80386–80486 Only Converts the signed doubleword in EAX to a signed quadword in the EDX:EAX register pair by extending the sign bit of EAX into all bits of EDX.

Flags No change

Encoding 10011001*

Syntax	Examples	CPU	Clock Cycles
CDQ	`cdq`	88/86	—
		286	—
		386	2
		486	3

* **CWD** and **CDQ** have the same encoding with two exceptions: in 32-bit mode, **CWD** is preceded by the operand-size byte (66h) but **CDQ** is not; in 16-bit mode, **CDQ** is preceded by the operand-size byte but **CWD** is not.

CLC Clear Carry Flag

Clears the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								0

Encoding 11111000

Syntax	Examples	CPU	Clock Cycles
CLC	`clc`	88/86	2
		286	2
		386	2
		486	2

CLD Clear Direction Flag

Clears the direction flag. All subsequent string instructions will process up (from low addresses to high addresses) by increasing the appropriate index registers.

Flags

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

Encoding 11111100

Syntax	Examples	CPU	Clock Cycles
CLD	`cld`	88/86	2
		286	2
		386	2
		486	2

CLI Clear Interrupt Flag

Clears the interrupt flag. When the interrupt flag is cleared, maskable interrupts are not recognized until the flag is set again with the **STI** instruction. In protected mode, **CLI** clears the flag only if the current task's privilege level is less than or equal to the value of the IOPL flag. Otherwise, a general-protection fault occurs.

Flags

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

Encoding 11111010

Syntax	Examples	CPU	Clock Cycles
CLI	`cli`	88/86	2
		286	3
		386	3
		486	5

CLTS Clear Task-Switched Flag

80286–80486 Privileged Only Clears the task-switched flag in the Machine Status Word (MSW) of the 80286, or the CR0 register of the 80386–80486. This instruction can be used only in system software executing at privilege level

0. See Intel documentation for details on the task-switched flag and other privileged-mode concepts.

Flags No change

Encoding 00001111 00000110

Syntax	Examples	CPU	Clock Cycles
CLTS	`clts`	88/86	—
		286	2
		386	5
		486	7

CMC Complement Carry Flag

Complements (toggles) the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								±

Encoding 11110101

Syntax	Examples	CPU	Clock Cycles
CMC	`cmc`	88/86	2
		286	2
		386	2
		486	2

CMP Compare Two Operands

Compares two operands as a test for a subsequent conditional-jump or set instruction. **CMP** does this by subtracting the source operand from the destination operand and setting the flags according to the result. **CMP** is the same as the **SUB** instruction, except that the result is not stored.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 001110*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *reg,reg*	`cmp di,bx`	88/86	3
	`cmp dl,cl`	286	2
		386	2
		486	1
CMP *mem,reg*	`cmp maximum,dx`	88/86	9+*EA*
	`cmp array[si],bl`	286	(*W88*=13+*EA*)
		386	7
		486	5
			2
CMP *reg,mem*	`cmp dx,minimum`	88/86	9+*EA*
	`cmp bh,array[si]`	286	(*W88*=13+*EA*)
		386	6
		486	6
			2

Encoding 100000*sw* *mod, 111,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *reg,immed*	`cmp bx,24`	88/86	4
		286	3
		386	2
		486	1
CMP *mem,immed*	`cmp WORD PTR [di],4`	88/86	10+*EA*
	`cmp tester,4000`	286	(*W88*=14+*EA*)
		386	6
		486	5
			2

Encoding 0011110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *accum,immed*	`cmp ax,1000`	88/86	4
		286	3
		386	2
		486	1

CMPS/CMPSB/CMPSW/CMPSD Compare String

Compares two strings. DS:SI must point to the source string and ES:DI must point to the destination string (even if operands are given). For each comparison, the destination element is subtracted from the source element and the flags are updated to reflect the result (although the result is not stored). DI and SI are adjusted according to the size of the operands and the status of the direction flag. They are increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **CMPS** form of the instruction is used, operands must be provided to indicate the size of the data elements to be processed. A segment override can be given for the source (but not for the destination). If **CMPSB** (bytes), **CMPSW** (words), or **CMPSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed.

CMPS and its variations are normally used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first match between two strings. **REPE** (or **REPZ**) is used to find the first mismatch. Before the comparison, CX should contain the maximum number of elements to compare. After a **REPNE CMPS**, the zero flag is clear if no match was found. After a **REPE CMPS**, the zero flag is set if no mismatch was found.

When the instruction finishes, ES:DI and DS:SI point to the element that follows (if the direction flag is clear) or precedes (if the direction flag is set) the match or mismatch. If CX decrements to 0, ES:DI and DS:SI point to the element that follows or precedes the last comparison. The zero flag is set or clear according to the result of the last comparison, not according to the value of CX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

1010011*w*

Syntax	Examples	CPU	Clock Cycles
CMPS ⟦*segreg***:**⟧ *src***,** ⟦**ES:**⟧ *dest*	`cmps source,es:dest`	88/86	22 (*W88*=30)
CMPSB ⟦⟦*segreg***:**⟦ *src***,** ⟧**ES:**⟧ *dest*⟧	`repne cmpsw`	286	8
CMPSW ⟦⟦*segreg***:**⟦ *src***,** ⟧**ES:**⟧ *dest*⟧	`repe cmpsb`	386	10
CMPSD ⟦⟦*segreg***:**⟦ *src***,** ⟧**ES:**⟧ *dest*⟧	`repne cmpsd`	486	8

CMPXCHG Compare and Exchange

80486 Only Compares the destination operand to the accumulator (AL, AX, or EAX). If equal, the source operand is copied to the destination. Otherwise, the destination is copied to the accumulator. The instruction sets flags according to the result of the comparison.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 00001111 1011000*b* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
CMPXCHG *mem,reg*	`cmpxchg  warr[bx],cx`	88/86	—
	`cmpxchg  string,bl`	286	—
		386	—
		486	7–10
CMPXCHG *reg,reg*	`cmpxchg  dl,cl`	88/86	—
	`cmpxchg  bx,dx`	286	—
		386	—
		486	6

CWD Convert Word to Double

Converts the signed word in AX to a signed doubleword in the DX:AX register pair by extending the sign bit of AX into all bits of DX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 10011001*

Syntax	Examples	CPU	Clock Cycles
CWD	`cwd`	88/86	5
		286	2
		386	2
		486	3

* **CWD** and **CDQ** have the same encoding with two exceptions: in 32-bit mode, **CWD** is preceded by the operand-size byte (66h) but **CDQ** is not; in 16-bit mode, **CDQ** is preceded by the operand-size byte but **CWD** is not.

CWDE Convert Word to Extended Double

80386–80486 Only Converts a signed word in AX to a signed doubleword in EAX by extending the sign bit of AX into all bits of EAX.

Flags No change

Encoding 10011000*

Syntax	Examples	CPU	Clock Cycles
CWDE	`cwde`	88/86	—
		286	—
		386	3
		486	3

* **CBW** and **CWDE** have the same encoding with two exceptions: in 32-bit mode, **CBW** is preceded by the operand-size byte (66h) but **CWDE** is not; in 16-bit mode, **CWDE** is preceded by the operand-size byte but **CBW** is not.

DAA Decimal Adjust After Addition

Adjusts the result of an addition to a packed BCD number (less than 100 decimal). The previous addition instruction should place its 8-bit binary sum in AL. **DAA** converts this binary sum to packed BCD format with the least significant decimal digit in the lower four bits and the most significant digit in the upper four bits. If the sum is greater than 99h after adjustment, the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Encoding 00100111

Syntax	Examples	CPU	Clock Cycles
DAA	`daa`	88/86	4
		286	3
		386	4
		486	2

DAS Decimal Adjust After Subtraction

Adjusts the result of a subtraction to a packed BCD number (less than 100 decimal). The previous subtraction instruction should place its 8-bit binary result in AL. **DAS** converts this binary sum to packed BCD format with the least significant decimal digit in the lower four bits and the most significant digit in the upper four bits. If the sum is greater than 99h after adjustment, the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Encoding

00101111

Syntax	Examples	CPU	Clock Cycles
DAS	`das`	88/86	4
		286	3
		386	4
		486	2

DEC Decrement

Subtracts 1 from the destination operand. Because the operand is treated as an unsigned integer, the **DEC** instruction does not affect the carry flag. To detect any effects on the carry flag, use the **SUB** instruction.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Encoding

1111111*w* *mod*, 001,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
DEC *reg8*	`dec cl`	88/86	3
		286	2
		386	2
		486	1
DEC *mem*	`dec counter`	88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	6
		486	3

Encoding 01001 *reg*

Syntax	Examples	CPU	Clock Cycles
DEC *reg16*	`dec ax`	88/86	3
DEC *reg32**		286	2
		386	2
		486	1

* 80386–80486 only.

DIV Unsigned Divide

Divides an implied destination operand by a specified source operand. Both operands are treated as unsigned numbers. If the source (divisor) is 16 bits wide, the implied destination (dividend) is the DX:AX register pair. The quotient goes into AX and the remainder into DX. If the source is 8 bits wide, the implied destination operand is AX. The quotient goes into AL and the remainder into AH. On the 80386–80486, if the source is EAX, the quotient goes into EAX and the remainder into EDX.

Flags

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

Encoding 1111011*w* *mod, 110,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
DIV *reg*	`div cx`	88/86	*b*=80–90,*w*=144–162
	`div dl`	286	*b*=14,*w*=22
		386	*b*=14,*w*=22,*d*=38
		486	*b*=16,*w*=24,*d*=40
DIV *mem*	`div [bx]`	88/86	(*b*=86–96,*w*=150–168)+*EA**
	`div fsize`		
		286	*b*=17,*w*=25
		386	*b*=17,*w*=25,*d*=41
		486	*b*=16,*w*=24,*d*=40

* Word memory operands on the 8088 take (158–176)+*EA* clocks.

ENTER Make Stack Frame

80286-80486 Only Creates a stack frame for a procedure that receives parameters passed on the stack. When *immed16* is 0, **ENTER** is equivalent to `push bp`, followed by `mov bp,sp`. The first operand of the **ENTER** instruction specifies the number of bytes to reserve for local variables. The second operand specifies the nesting level for the procedure. The nesting level should be 0 for languages that do not allow access to local variables of higher-level procedures (such as C, Basic, and FORTRAN). See the complementary instruction **LEAVE** for a method of exiting from a procedure.

Flags No change

Encoding 11001000 *data (2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
ENTER *immed16*,**0**	`enter 4,0`	88/86	—
		286	11
		386	10
		486	14
ENTER *immed16*,**1**	`enter 0,1`	88/86	—
		286	15
		386	12
		486	17
ENTER *immed16,immed8*	`enter 6,4`	88/86	—
		286	12+4(n – 1)
		386	15+4(n – 1)
		486	17+3n

HLT Halt

Stops CPU execution until an interrupt restarts execution at the instruction following **HLT**. In protected mode, this instruction works only in privileged mode.

Flags No change

Encoding

11110100

Syntax	Examples	CPU	Clock Cycles
HLT	`hlt`	88/86	2
		286	2
		386	5
		486	4

IDIV Signed Divide

Divides an implied destination operand by a specified source operand. Both operands are treated as signed numbers. If the source (divisor) is 16 bits wide, the implied destination (dividend) is the DX:AX register pair. The quotient goes into AX and the remainder into DX. If the source is 8 bits wide, the implied destination is AX. The quotient goes into AL and the remainder into AH. On the 80386–80486, if the source is EAX, the quotient goes into EAX and the remainder into EDX.

Flags

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

Encoding 1111011*w* *mod, 111,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles

IDIV *reg*	`idiv bx`
	`idiv dl`
		88/86	*b*=101–112,*w*=165–184

		286	*b*=17,*w*=25
		386	*b*=19,*w*=27,*d*=43
		486	*b*=19,*w*=27,*d*=43
IDIV *mem*	`idiv itemp`	88/86	(*b*=107–118,*w*=171–190)+*EA**
		286	*b*=20,*w*=28
		386	*b*=22,*w*=30,*d*=46
		486	*b*=20,*w*=28,*d*=44

* Word memory operands on the 8088 take (175–194)+*EA* clocks.

IMUL Signed Multiply

Multiplies an implied destination operand by a specified source operand. Both operands are treated as signed numbers. If a single 16-bit operand is given, the implied destination is AX and the product goes into the DX:AX register pair. If a single 8-bit operand is given, the implied destination is AL and the product goes into AX. On the 80386–80486, if the operand is EAX, the product goes into the EDX:EAX register pair. The carry and overflow flags are set if the product is sign-extended into DX for 16-bit operands, into AH for 8-bit operands, or into EDX for 32-bit operands.

Two additional syntaxes are available on the 80186–80486 processors. In the two-operand form, a 16-bit register gives one of the factors and serves as the destination for the result; a source constant specifies the other factor. In the three-operand form, the first operand is a 16-bit register where the result will be stored, the second is a 16-bit register or memory operand containing one of the factors, and the third is a constant representing the other factor. With both variations, the overflow and carry flags are set if the result is too large to fit into the 16-bit destination register. Since the low 16 bits of the product are the same for both signed and unsigned multiplication, these syntaxes can be used for either signed or unsigned numbers. On the 80386–80486, the operands can be either 16 or 32 bits wide.

A fourth syntax is available on the 80386–80486. Both the source and destination operands can be given specifically. The source can be any 16- or 32-bit memory operand or general-purpose register. The destination can be any general-purpose register of the same size. The overflow and carry flags are set if the product does not fit in the destination.

Flags

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Encoding

1111011*w* *mod, 101,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg*	`imul dx`	88/86	*b*=80–98,*w*=128–154
		286	*b*=13,*w*=21
		386	*b*=9–14,*w*=9–22,*d*=9–38*
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *mem*	`imul factor`	88/86	(*b*=86–104,*w*=134–160)+*EA*†
		286	*b*=16,*w*=24
		386	*b*=12–17,*w*=12–25,*d*=12–41*
		486	*b*=13–18,*w*=13–26, *d*=13–42

* The 80386–80486 processors have an early-out multiplication algorithm. Therefore, multiplying an 8-bit or 16-bit value in EAX takes the same time as multiplying the value in AL or AX.

† Word memory operands on the 8088 take (138–164)+*EA* clocks.

Encoding

011010*s*1 *mod, reg, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,immed*	`imul cx,25`	88/86	—
IMUL *reg32,immed**		286	21
		386	*b*=9–14,*w*=9–22,*d*=9–38†
		486	*b*=13–18,*w*=13–26,*d*=13–42

IMUL *reg16*,*reg16*,*immed*	`imul`	88/86	—
IMUL *reg32*,*reg32*,*immed**	`dx,ax,18`	286	21
		386	*b*=9–14,*w*=9–22,*d*=9–38†
		486	*b*=13–18,*w*=13–26,*d*=13–42

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,mem16,immed*	`imul`	88/86	—
IMUL *reg32,mem32,immed**	`bx,[si],60`	286	24
		386	*b*=12–17,*w*=12–25,*d*=12–41†
		486	*b*=13–18,*w*=13–26,*d*=13–42

Encoding 00001111 10101111 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,reg16*	`imul cx,ax`	88/86	—
IMUL *reg32,reg32**		286	—
		386	*w*=9–22,*d*=9–38
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *reg16,mem16*	`imul`	88/86	—
IMUL *reg32,mem32**	`dx,[si]`	286	—
		386	*w*=12–25,*d*=12–41
		486	*b*=13–18,*w*=13–26,*d*=13–42

* 80386–80486 only.

† The variations depend on the source constant size; destination size is not a factor.

IN Input from Port

Transfers a byte or word (or doubleword on the 80386–80486) from a port to the accumulator register. The port address is specified by the source operand, which can be DX or an 8-bit constant. Constants can be used only for port numbers less than 255; use DX for higher port numbers. In protected mode, a general-protection fault occurs if **IN** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 1110010*w* *data (1)*

Syntax	Examples	CPU	Clock Cycles
IN *accum,immed*	`in ax,60h`	88/86	10 (*W88*=14)
		286	5
		386	12,*pm*=6,26*
		486	14,*pm*=9,29*†

Encoding 1110110*w*

Syntax	Examples	CPU	Clock Cycles
IN *accum*,**DX**	`in ax,dx`	88/86	8 (*W88*=12)
	`in al,dx`	286	5
		386	13,*pm*=7,27*
		486	14,*pm*=8,28*†

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

† Takes 27 clocks in virtual 8086 mode.

INC Increment

Adds 1 to the destination operand. Because the operand is treated as an unsigned integer, the **INC** instruction does not affect the carry flag. If a signed carry requires detection, use the **ADD** instruction.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Encoding 1111111*w* *mod,000,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
INC *reg8*	`inc cl`	88/86	3
		286	2
		386	2
		486	1
INC *mem*	`inc vpage`	88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	6
		486	3

Encoding 01000 *reg*

Syntax	Examples	CPU	Clock Cycles
INC *reg16*	`inc bx`	88/86	3
INC *reg32**		286	2
		386	2
		486	1

* 80386–80486 only.

INS/INSB/INSW/INSD Input from Port to String

80286-80486 Only Receives a string from a port. The string is considered the destination and must be pointed to by ES:DI (even if an operand is given). The input port is specified in DX. For each element received, DI is adjusted according to the size of the operand and the status of the direction flag. DI is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **INS** form of the instruction is used, a destination operand must be provided to indicate the size of the data elements to be processed, and DX must be specified as the source operand containing the port number. A segment override is not allowed. If **INSB** (bytes), **INSW** (words), or **INSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be received.

INS and its variations are normally used with the **REP** prefix. Before the repeated instruction is executed, CX should contain the number of elements to be received. In protected mode, a general-protection fault occurs if **INS** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 0110110*w*

Syntax	Examples	CPU	Clock Cycles
INS ⟦**ES:**⟧ *dest*, **DX**	`ins es:instr,dx`	88/86	—
INSB ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insb`	286	5
INSW ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insw`	386	15,*pm*=9,29*
INSD ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insd`	486	17,*pm*=10,32*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

INT Interrupt

Generates a software interrupt. An 8-bit constant operand (0 to 255) specifies the interrupt procedure to be called. The call is made by indexing the interrupt number into the Interrupt Vector Table (IVT) starting at segment 0, offset 0. In real mode, the IVT contains 4-byte pointers to interrupt procedures. In privileged mode, the IVT contains 8-byte pointers.

When an interrupt is called in real mode, the flags, CS, and IP are pushed onto the stack (in that order), and the trap and interrupt flags are cleared. **STI** can be used to restore interrupts. See Intel documentation and the documentation for your operating system for details on using and defining interrupts in privileged mode. To return from an interrupt, use the **IRET** instruction.

Flags

O	D	I	T	S	Z	A	P	C
		0	0					

Encoding 11001101 *data (1)*

Syntax	Examples	CPU	Clock Cycles
INT *immed8*	`int 25h`	88/86	51 (*88*=71)
		286	23+*m*,*pm*=(40,78)+*m**
		386	37,*pm*=59,99*
		486	30,*pm*=44,71*

Encoding 11001100

Syntax	Examples	CPU	Clock Cycles
INT 3	`int 3`	88/86	52 (*88*=72)
		286	23+*m*,*pm*=(40,78)+*m**
		386	33,*pm*=59,99*
		486	26,*pm*=44,71*

* The first protected-mode timing is for interrupts to the same privilege level. The second is for interrupts to a higher privilege level. Timings for interrupts through task gates are not shown.

INTO Interrupt on Overflow

Generates Interrupt 4 if the overflow flag is set. The default MS-DOS behavior for Interrupt 4 is to return without taking any action. For **INTO** to have any effect, you must define an interrupt procedure for Interrupt 4.

Flags

O	D	I	T	S	Z	A	P	C
		±	±					

Encoding 11001110

Syntax	Examples	CPU	Clock Cycles
INTO	`into`	88/86	53 (*88*=73),*noj*=4
		286	24+*m*,*noj*=3,*pm*=(40,78)+*m**
		386	35,*noj*=3,*pm*=59,99*
		486	28,*noj*=3,*pm*=46,73*

* The first protected-mode timing is for interrupts to the same privilege level. The second is for interrupts to a higher privilege level. Timings for interrupts through task gates are not shown.

INVD Invalidate Data Cache

80486 Only Empties contents of the current data cache without writing changes to memory. Proper use of this instruction requires knowledge of how contents are placed in the cache. **INVD** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00001000

Syntax	Examples	CPU	Clock Cycles
INVD	`invd`	88/86	—
		286	—
		386	—
		486	4

INVLPG Invalidate TLB Entry

80486 Only Invalidates an entry in the Translation Lookaside Buffer (TLB), used by the demand-paging mechanism in virtual-memory operating systems. The instruction takes a single memory operand and calculates the effective address of the operand, including the segment address. If the resulting address is mapped by any entry in the TLB, this entry is removed. Proper use of **INVLPG** requires understanding the hardware-supported demand-paging mechanism. **INVLPG** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00000001 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
INVLPG	`invlpg  pointer[bx]`	88/86	—
	`invlpg  es:entry`	286	—
		386	—
		486	12*

* 11 clocks if address is not mapped by any TLB entry.

IRET/IRETD Interrupt Return

Returns control from an interrupt procedure to the interrupted code. In real mode, the **IRET** instruction pops IP, CS, and the flags (in that order) and resumes execution. See Intel documentation for details on **IRET** operation in privileged mode. On the 80386–80486, the **IRETD** instruction should be used to pop a 32-bit instruction pointer when returning from an interrupt called from a 32-bit segment. The **F** suffix prevents epilogue code from being generated when ending a **PROC** block. Use it to terminate interrupt service procedures.

Flags

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

Encoding

11001111

Syntax	Examples	CPU	Clock Cycles
IRET	`iret`	88/86	32 (*88*=44)
IRETD*		286	17+*m*,*pm*=(31,55)+*m*†
IRETF		386	22,*pm*=38,82†
IRETDF*		486	15,*pm*=20,36

* 80386–80486 only.

† The first protected-mode timing is for interrupts to the same privilege level within a task. The second is for interrupts to a higher privilege level within a task. Timings for interrupts through task gates are not shown.

J*condition* Jump Conditionally

Transfers execution to the specified label if the flags condition is true. The *condition* is tested by checking the flags shown in the table on the following page. If *condition* is false, no jump is taken and program execution continues at the next instruction. On the 8086–80286 processors, the label given as the operand must be short (between –128 and +127 bytes from the instruction following the jump).* The 80386–80486 processors allow near jumps (–32,768 to +32,767 bytes). On the 80386–80486, the assembler generates the shortest jump possible, unless the jump size is explicitly specified.

When the 80386–80486 processors are in **FLAT** memory model, short jumps range from –128 to +127 bytes and near jumps range from –2 to +2 gigabytes. There are no far jumps.

Flags

No change

Encoding

0111*cond* *disp (1)*

Syntax	Examples	CPU	Clock Cycles
J*condition label*	`jg bigger`	88/86	16,*noj*=4
	`jo SHORT too_big`	286	7+*m*,*noj*=3
	`jpe p_even`	386	7+*m*,*noj*=3
		486	3,*noj*=1

Encoding

00001111 1000*cond* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
J*condition label*†	`je next`	88/86	—
	`jnae lesser`	286	—
	`js negative`	386	7+*m*,*noj*=3
		486	3,*noj*=1

* If a source file for an 8086–80286 program contains a conditional jump outside the range of –128 to +127 bytes, the assembler emits a level 3 warning and generates two instructions (including an unconditional jump) that are the equivalent of the desired instruction. This behavior can be enabled and disabled with the **OPTION LJMP** and **OPTION NOLJMP** directives.

† Near labels are only available on the 80386–80486. They are the default.

Jump Conditions

Opcode*	Mnemonic	Flags Checked	Description
size 0010	**JB/JNAE**	CF=1	Jump if below/not above or equal (unsigned comparisons)
size 0011	**JAE/JNB**	CF=0	Jump if above or equal/not below (unsigned comparisons)
size 0110	**JBE/JNA**	CF=1 or ZF=1	Jump if below or equal/not above (unsigned comparisons)
size 0111	**JA/JNBE**	CF=0 and ZF=0	Jump if above/not below or equal (unsigned comparisons)
size 0100	**JE/JZ**	ZF=1	Jump if equal (zero)
size 0101	**JNE/JNZ**	ZF=0	Jump if not equal (not zero)
size 1100	**JL/JNGE**	SF_OF	Jump if less/not greater or equal (signed comparisons)
size 1101	**JGE/JNL**	SF=OF	Jump if greater or equal/not less (signed comparisons)
size 1110	**JLE/JNG**	ZF=1 or SF_OF	Jump if less or equal/not greater (signed comparisons)
size 1111	**JG/JNLE**	ZF=0 and SF=OF	Jump if greater/not less or equal (signed comparisons)
size 1000	**JS**	SF=1	Jump if sign
size 1001	**JNS**	SF=0	Jump if not sign

Opcode*	Mnemonic	Flags Checked	Description
size 0010	**JC**	CF=1	Jump if carry
size 0011	**JNC**	CF=0	Jump if not carry
size 0000	**JO**	OF=1	Jump if overflow
size 0001	**JNO**	OF=0	Jump if not overflow
size 1010	**JP/JPE**	PF=1	Jump if parity/parity even
size 1011	**JNP/JPO**	PF=0	Jump if no parity/parity odd

* The *size* bits are 0111 for short jumps or 1000 for 80386–80486 near jumps.

JCXZ/JECXZ Jump if CX is Zero

Transfers program execution to the specified label if CX is 0. On the 80386–80486, **JECXZ** can be used to jump if ECX is 0. If the count register is not 0, execution continues at the next instruction. The label given as the operand must be short (between –128 and +127 bytes from the instruction following the jump).

Flags No change

Encoding 11100011 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
JCXZ *label*	`jcxz  not found`	88/86	18,*noj*=6
JECXZ *label**		286	8+*m*,*noj*=4
		386	9+*m*,*noj*=5
		486	8,*noj*=5

* 80386–80486 only.

JMP Jump Unconditionally

Transfers program execution to the address specified by the destination operand. Jumps are near (between –32,768 and +32,767 bytes from the instruction following the jump), or short (between –128 and +127 bytes), or far (in a different code segment). Unless a distance is explicitly specified, the assembler selects the shortest possible jump. With near and short jumps, the operand specifies a new IP address. With far jumps, the operand specifies new IP and CS addresses.

When the 80386–80486 processors are in **FLAT** memory model, short jumps range from –128 to +127 bytes and near jumps range from –2 to +2 gigabytes.

Flags No change

Encoding 11101011 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp SHORT exit`	88/86	15
		286	7+*m*
		386	7+*m*
		486	3

Encoding 11101001 *disp (2*)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp close`	88/86	15
	`jmp NEAR PTR distant`	286	7+*m*
		386	7+*m*
		486	3

Encoding 11101010 *disp (4*)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp FAR PTR close`	88/86	15
	`jmp distant`	286	11+*m*,*pm*=23+*m*†
		386	12+*m*,*pm*=27+*m*†
		486	17,*pm*=19†

Encoding 11111111 *mod*,100,*r/m* *disp (0 or 2)*

Syntax	Examples	CPU	Clock Cycles
JMP *reg16*	`jmp ax`	88/86	11
JMP *mem32*§		286	7+*m*
		386	7+*m*
		486	5
JMP *mem16*	`jmp WORD PTR [bx]`	88/86	18+*EA*
JMP *mem32*§	`jmp table[di]`	286	11+*m*
	`jmp DWORD PTR [si]`	386	10+*m*
		486	5

Encoding 11111111 *mod*,101,*r/m* *disp (4*)*

Syntax	Examples	CPU	Clock Cycles
JMP *mem32*	`jmp fpointer[si]`	88/86	24+*EA*
JMP *mem48*§	`jmp DWORD PTR [bx]`	286	15+*m*,*pm*=26+*m*
	`jmp FWORD PTR [di]`	386	12+*m*,*pm*=27+*m*
		486	13,*pm*=18

* On the 80386–80486, the displacement can be 4 bytes for near jumps or 6 bytes for far jumps.

† Timings for jumps through call or task gates are not shown, since they are normally used only in operating systems.

§ 80386–80486 only. You can use **DWORD PTR** to specify near register-indirect jumps or **FWORD PTR** to specify far register-indirect jumps.

LAHF Load Flags into AH Register

Transfers bits 0 to 7 of the flags register to AH. This includes the carry, parity, auxiliary carry, zero, and sign flags, but not the trap, interrupt, direction, or overflow flags.

Flags No change

Encoding 10011111

Syntax	Examples	CPU	Clock Cycles
LAHF	`lahf`	88/86	4
		286	2
		386	2
		486	3

LAR Load Access Rights

80286-80486 Protected Only Loads the access rights of a selector into a specified register. The source operand must be a register or memory operand containing a selector. The destination operand must be a register that will receive the access rights if the selector is valid and visible at the current privilege level. The zero flag is set if the access rights are transferred, or cleared if they are not. See Intel documentation for details on selectors, access rights, and other privileged-mode concepts.

Flags

```
O D I  T  S  Z  A  P  C
               ±
```

Encoding 00001111 00000010 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
LAR *reg16,reg16*	`lar ax,bx`	88/86	—
LAR *reg32,reg32**		286	14
		386	15
		486	11
LAR *reg16,mem16*	`lar cx,selector`	88/86	—
LAR *reg32,mem32**		286	16
		386	16
		486	11

* 80386–80486 only.

LDS/LES/LFS/LGS/LSS Load Far Pointer

Reads and stores the far pointer specified by the source memory operand. The instruction moves the pointer's segment value into DS, ES, FS, GS, or SS (depending on the instruction). Then it moves the pointer's offset value into the destination operand. The **LDS** and **LES** instructions are available on all processors. The **LFS**, **LGS**, and **LSS** instructions are available only on the 80386–80486.

Flags No change

Encoding 11000101 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LDS *reg,mem*	`lds si,fpointer`	88/86	16+*EA* (*88*=24+*EA*)
		286	7,*pm*=21
		386	7,*pm*=22
		486	6,*pm*=12

Encoding 11000100 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LES *reg,mem*	`les di,fpointer`	88/86	16+*EA* (*88*=24+*EA*)
		286	7,*pm*=21
		386	7,*pm*=22
		486	6,*pm*=12

Encoding 00001111 10110100 *mod, reg, r/m* *disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LFS *reg,mem*	`lfs edi,fpointer`	88/86	—
		286	—
		386	7,*pm*=25
		486	6,*pm*=12

Encoding 00001111 10110101 *mod, reg, r/m* *disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LGS *reg,mem*	`lgs bx,fpointer`	88/86	—
		286	—
		386	7,*pm*=25
		486	6,*pm*=12

Encoding 00001111 10110010 *mod, reg, r/m* *disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LSS *reg,mem*	`lss bp,fpointer`	88/86	—
		286	—
		386	7,*pm*=22
		486	6,*pm*=12

LEA Load Effective Address

Calculates the effective address (offset) of the source memory operand and stores the result in the destination register. If the source operand is a direct memory address, the assembler encodes the instruction in the more efficient `MOV reg,immediate` form (equivalent to **MOV** *reg***, OFFSET** *mem*).

Flags No change

Encoding 10001101 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LEA *reg16,mem*	`lea bx,npointer`	88/86	2+*EA*
LEA *reg32,mem**		286	3
		386	2
		486	1†

* 80386–80486 only.

† 2 if index register used.

LEAVE High Level Procedure Exit

Terminates the stack frame of a procedure. **LEAVE** reverses the action of a previous **ENTER** instruction by restoring SP and BP to the values they had before the procedure stack frame was initialized. **LEAVE** is equivalent to `mov sp,bp`, followed by `pop bp`.

Flags No change

Encoding 11001001

Syntax	Examples	CPU	Clock Cycles
LEAVE	`leave`	88/86	—
		286	5
		386	4
		486	5

LES/LFS/LGS Load Far Pointer to Extra Segment

See **LDS**.

LGDT/LIDT/LLDT Load Descriptor Table

Loads a value from an operand into a descriptor table register. **LGDT** loads into the Global Descriptor Table, **LIDT** into the Interrupt Vector Table, and **LLDT** into the Local Descriptor Table. These instructions are available only in privileged mode. See Intel documentation for details on descriptor tables and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod, 010,r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LGDT *mem48*	`lgdt descriptor`	88/86	—
		286	11
		386	11
		486	11

Encoding 00001111 00000001 *mod, 011,r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LIDT *mem48*	`lidt descriptor`	88/86	—
		286	12
		386	11
		486	11

Encoding 00001111 00000000 *mod, 010,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LLDT *reg16*	`lldt ax`	88/86	—
		286	17
		386	20
		486	11
LLDT *mem16*	`lldt selector`	88/86	—
		286	19
		386	24
		486	11

LMSW Load Machine Status Word

80286-80486 Privileged Only Loads a value from a memory operand into the Machine Status Word (MSW). This instruction is available only in privileged mode. See Intel documentation for details on the MSW and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod, 110,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LMSW *reg16*	`lmsw ax`	88/86	—
		286	3
		386	10
		486	13
LMSW *mem16*	`lmsw machine`	88/86	—
		286	6
		386	13
		486	13

LOCK Lock the Bus

Locks out other processors during execution of the next instruction. This instruction is a prefix. It must precede an instruction that accesses a memory location that another processor might attempt to access at the same time. See Intel documentation for details on multiprocessor environments.

Flags No change

Encoding 11110000

Syntax	Examples	CPU	Clock Cycles
LOCK *instruction*	`lock xchg ax,sem`	88/86	2
		286	0
		386	0
		486	1

LODS/LODSB/LODSW/LODSD Load Accumulator from String

Loads the accumulator register with an element from a string in memory. DS:SI must point to the source element, even if an operand is given. For each source element loaded, SI is adjusted according to the size of the operand and the status of the direction flag. SI is incremented if the direction flag has been cleared with **CLD** or decremented if the direction flag has been set with **STD**.

If the **LODS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. A segment override can be given. If **LODSB** (bytes), **LODSW** (words), or **LODSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element will be loaded to AL, AX, or EAX.

LODS and its variations are not used with repeat prefixes, since there is no reason to repeatedly load memory values to a register.

Flags No change

Encoding 1010110*w*

Syntax	**Examples**	**CPU**	**Clock Cycles**
LODS ⟦*segreg*:⟧*src*	`lods  es:source`	88/86	12 (*W88*=16)
LODSB ⟦⟦*segreg*:⟧*src*⟧	`lodsw`	286	5
LODSW⟦⟦*segreg*:⟧*src*⟧		386	5
LODSD ⟦⟦*segreg*:⟧*src*⟧		486	5

LOOP/LOOPW/LOOPD Loop

Loops repeatedly to a specified label. **LOOP** decrements CX (without changing any flags) and, if the result is not 0, transfers execution to the address specified by the operand. On the 80386–80486, **LOOP** uses the 16-bit CX in 16-bit mode and the 32-bit ECX in 32-bit mode. The default can be overridden with **LOOPW** (CX) or **LOOPD** (ECX). If CX is 0 after being decremented, execution continues at the next instruction. The operand must specify a short label (between –128 and +127 bytes from the instruction following the **LOOP** instruction).

Flags No change

Encoding 11100010 *disp (1)*

Syntax	**Examples**	**CPU**	**Clock Cycles**
LOOP *label*	`loop  wend`	88/86	17,*noj*=5
LOOPW *label**		286	8+*m*,*noj*=4
LOOPD *label**		386	11+*m*
		486	7,*noj*=6

* 80386–80486 only.

LOOP*condition*/LOOP*condition*W/LOOP*condition*D Loop Conditionally

Loops repeatedly to a specified label if *condition* is met and if CX is not 0. On the 80386–80486, these instructions use the 16-bit CX in 16-bit mode and the 32-bit ECX in 32-bit mode. This default can be overridden with the **W** (CX) or **D** (ECX) forms of the instruction. The instruction decrements CX (without changing any flags) and tests whether the zero flag was set by a previous instruction (such as **CMP**). With **LOOPE** and **LOOPZ** (they are synonyms), execution is transferred to the label if the zero flag is set and CX is not 0. With **LOOPNE** and **LOOPNZ** (they are synonyms), execution is transferred to the label if the zero flag is cleared and CX is not 0. Execution continues at the next instruction if the condition is not met. Before entering the loop, CX should be set to the maximum number of repetitions desired.

Flags No change

Encoding 11100001 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOPE *label*	`loopz again`	88/86	18,*noj*=6
LOOPEW *label**		286	8+*m*,*noj*=4
LOOPED *label**		386	11+*m*
LOOPZ *label*		486	9,*noj*=6
LOOPZW *label**			
LOOPZD *label**			

Encoding 11100000 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOPNE *label*	`loopnz for_next`	88/86	19,*noj*=5
LOOPNEW *label**		286	8,*noj*=4
LOOPNED *label**		386	11+*m*
LOOPNZ *label*		486	9,*noj*=6
LOOPNZW *label**			
LOOPNZD *label**			

* 80386–80486 only.

LSL Load Segment Limit

80286-80486 Protected Only Loads the segment limit of a selector into a specified register. The source operand must be a register or memory operand containing a selector. The destination operand must be a register that will receive the segment limit if the selector is valid and visible at the current privilege level. The zero flag is set if the segment limit is transferred, or cleared if it is not. See Intel documentation for details on selectors, segment limits, and other protected-mode concepts.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 00000011 *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LSL *reg16,reg16*	`lsl ax,bx`	88/86	—
LSL *reg32,reg32**		286	14
		386	20,25†
		486	10
LSL *reg16,mem16*	`lsl cx,seg_lim`	88/86	—
LSL *reg32,mem32**		286	16
		386	21,26†
		486	10

* 80386–80486 only.

† The first value is for byte granular; the second is for page granular.

LSS Load Far Pointer to Stack Segment

See **LDS**.

LTR Load Task Register

80286-80486 Protected Only Loads a value from the specified operand to the current task register. **LTR** is available only in privileged mode. See Intel documentation for details on task registers and other protected-mode concepts.

Flags No change

Encoding 00001111 00000000 *mod, 011,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LTR *reg16*	`ltr ax`	88/86	—
		286	17
		386	23
		486	20
LTR *mem16*	`ltr task`	88/86	—
		286	19
		386	27
		486	20

MOV Move Data

Moves the value in the source operand to the destination operand. If the destination operand is SS, interrupts are disabled until the next instruction is executed (except on early versions of the 8088 and 8086).

Flags

No change

Encoding

100010*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *reg,reg*	`mov dh,bh`	88/86	2
	`mov dx,cx`	286	2
	`mov bp,sp`	386	2
		486	1
MOV *mem,reg*	`mov array[di],bx`	88/86	9+*EA* (*W88*=13+*EA*)
	`mov count,cx`	286	3
		386	2
		486	1
MOV *reg,mem*	`mov bx,pointer`	88/86	8+*EA* (*W88*=12+*EA*)
	`mov dx,matrix[bx+di]`	286	5
		386	4
		486	1

Encoding

1100011*w* *mod*, 000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *mem,immed*	`mov [bx],15`	88/86	10+*EA* (*W88*=14+*EA*)
	`mov color,7`	286	3
		386	2
		486	1

Encoding

1011*w reg* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *reg,immed*	`mov cx,256`	88/86	4
	`mov dx,OFFSET string`	286	2
		386	2
		486	1

Encoding 101000*aw* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
MOV *mem,accum*	`mov total,ax`	88/86	10 (*W88*=14)
		286	3
		386	2
		486	1
MOV *accum,mem*	`mov al,string`	88/86	10 (*W88*=14)
		286	5
		386	4
		486	1

Encoding 100011*d*0 *mod,sreg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *segreg,reg16*	`mov ds,ax`	88/86	2
		286	2,*pm*=17
		386	2,*pm*=18
		486	3,*pm*=9
MOV *segreg,mem16*	`mov es,psp`	88/86	8+*EA* (*88*=12+*EA*)
		286	5,*pm*=19
		386	5,*pm*=19
		486	3,*pm*=9
MOV *reg16,segreg*	`mov ax,ds`	88/86	2
		286	2
		386	2
		486	3
MOV *mem16,segreg*	`mov stack_save,ss`	88/86	9+*EA* (*88*=13+*EA*)
		286	3
		386	2
		486	3

MOV Move to/from Special Registers

80386–80486 Only Moves a value from a special register to or from a 32-bit general-purpose register. The special registers include the control registers CR0, CR2, and CR3; the debug registers DR0, DR1, DR2, DR3, DR6, and DR7; and the test registers TR6 and TR7. On the 80486, the test registers TR3, TR4, and TR5 are also available. See Intel documentation for details on special registers.

Flags

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

Encoding

00001111 001000*d*0 11, *reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32, controlreg*	`mov eax,cr2`	88/86	—
		286	—
		386	6
		486	4
MOV *controlreg,reg32*	`mov cr0,ebx`	88/86	—
		286	—
		386	CR0=10,CR2=4,CR3=5
		486	4,CR0=16

Encoding

00001111 001000*d*1 11, *reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32,debugreg*	`mov edx,dr3`	88/86	—
		286	—
		386	DR0–3=22,DR6–7=14
		486	10
MOV *debugreg,reg32*	`mov dr0,ecx`	88/86	—
		286	—
		386	DR0–3=22,DR6–7=16
		486	11

Encoding

00001111 001001*d*0 11,*reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32,testreg*	`mov edx,tr6`	88/86	—
		286	—
		386	12
		486	4,TR3=3
MOV *testreg, reg32*	`mov tr7,eax`	88/86	—
		286	—
		386	12
		486	4,TR3=6

* The *reg* field contains the register number of the special register (for example, 000 for CR0, 011 for DR7, or 111 for TR7).

MOVS/MOVSB/MOVSW/MOVSD Move String Data

Moves a string from one area of memory to another. DS:SI must point to the source string and ES:DI to the destination address, even if operands are given. For each element moved, DI and SI are adjusted according to the size of the operands and the status of the direction flag. They are increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **MOVS** form of the instruction is used, operands must be provided to indicate the size of the data elements to be processed. A segment override can be given for the source operand (but not for the destination). If **MOVSB** (bytes), **MOVSW** (words), or **MOVSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed.

MOVS and its variations are normally used with the **REP** prefix.

Flags No change

Encoding 1010010*w*

Syntax	Examples	CPU	Clock Cycles
MOVS ⟦**ES:**⟧*dest*,⟦*segreg*:⟧*src*	`rep   movsb`	88/86	18 (*W88*=26)
MOVSB ⟦⟦**ES:**⟧*dest*,⟦*segreg*:⟧*src*⟧	`movs  dest,es:source`	286	5
MOVSW ⟦⟦**ES:**⟧*dest*,⟦*segreg*:⟧*src*⟧		386	7
MOVSD ⟦⟦**ES:**⟧*dest*,⟦*segreg*:⟧*src*⟧		486	7

MOVSX Move with Sign-Extend

80386–80486 Only Moves and sign-extends the value of the source operand to the destination register. **MOVSX** is used to copy a signed 8-bit or 16-bit source operand to a larger 16-bit or 32-bit destination register.

Flags No change

Encoding 00001111 1011111*w* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
MOVSX *reg,reg*	`movsx   eax,bx`	88/86	—
	`movsx   ecx,bl`	286	—
	`movsx   bx,al`	386	3
		486	3

Syntax	Examples	CPU	Clock Cycles
MOVSX *reg,mem*	`movsx cx,bsign`	88/86	—
	`movsx edx,wsign`	286	—
	`movsx eax,bsign`	386	6
		486	3

MOVZX Move with Zero-Extend

80386–80486 Only Moves and zero-extends the value of the source operand to the destination register. **MOVZX** is used to copy an unsigned 8-bit or 16-bit source operand to a larger 16-bit or 32-bit destination register.

Flags No change

Encoding 00001111 1011011*w* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
MOVZX *reg,reg*	`movzx eax,bx`	88/86	—
	`movzx ecx,bl`	286	—
	`movzx bx,al`	386	3
		486	3
MOVZX *reg,mem*	`movzx cx,bunsign`	88/86	—
	`movzx edx,wunsign`	286	—
	`movzx eax,bunsign`	386	6
		486	3

MUL Unsigned Multiply

Multiplies an implied destination operand by a specified source operand. Both operands are treated as unsigned numbers. If a single 16-bit operand is given, the implied destination is AX and the product goes into the DX:AX register pair. If a single 8-bit operand is given, the implied destination is AL and the product goes into AX. On the 80386–80486, if the operand is EAX, the product goes into the EDX:EAX register pair. The carry and overflow flags are set if DX is not 0 for 16-bit operands or if AH is not 0 for 8-bit operands.

Flags

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Encoding 1111011*w* *mod*, 100, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MUL *reg*	`mul bx`	88/86	*b*=70–77,*w*=118–133
	`mul dl`	286	*b*=13,*w*=21
		386	*b*=9–14,*w*=9–22,*d*=9–38*
		486	*b*=13–18,*w*=13–26,*d*=13–42
MUL *mem*	`mul factor`	88/86	(*b*=76–83,*w*=124–139)+*EA*†
	`mul WORD PTR [bx]`	286	*b*=16,*w*=24
		386	*b*=12–17,*w*=12–25,*d*=12–41*
		486	*b*=13–18,*w*=13–26,*d*=13–42

* The 80386–80486 processors have an early-out multiplication algorithm. Therefore, multiplying an 8-bit or 16-bit value in EAX takes the same time as multiplying the value in AL or AX.

† Word memory operands on the 8088 take (128–143)+*EA* clocks.

NEG Two's Complement Negation

Replaces the operand with its two's complement. **NEG** does this by subtracting the operand from 0. If the operand is 0, the carry flag is cleared. Otherwise, the carry flag is set. If the operand contains the maximum possible negative value (–128 for 8-bit operands or –32,768 for 16-bit operands), the value does not change, but the overflow and carry flags are set.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1111011*w* *mod*, 011, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
NEG *reg*	`neg ax`	88/86	3
		286	2
		386	2
		486	1
NEG *mem*	`neg balance`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3

NOP No Operation

Performs no operation. **NOP** can be used for timing delays or alignment.

Flags No change

Encoding 10010000*

Syntax	Examples	CPU	Clock Cycles
NOP	`nop`	88/86	3
		286	3
		386	3
		486	3

* The encoding is the same as **XCHG AX,AX**.

NOT One's Complement Negation

Toggles each bit of the operand by clearing set bits and setting cleared bits.

Flags No change

Encoding 1111011*w* *mod*, 010, *r/m* *disp (0,1,or2)*

Syntax	Examples	CPU	Clock Cycles
NOT *reg*	`not ax`	88/86	3
		286	2
		386	2
		486	1
NOT *mem*	`not masker`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3

OR Inclusive OR

Performs a bitwise OR operation on the source and destination operands and stores the result to the destination operand. For each bit position in the operands, if either or both bits are set, the corresponding bit of the result is set. Otherwise, the corresponding bit of the result is cleared.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding

000010*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *reg,reg*	`or ax,dx`	88/86	3
		286	2
		386	2
		486	1
OR *mem,reg*	`or bits,dx`	88/86	16+*EA* (*W88*=24+*EA*)
	`or [bp+6],cx`	286	7
		386	7
		486	3
OR *reg,mem*	`or bx,masker`	88/86	9+*EA* (*W88*=13+*EA*)
	`or dx,color[di]`	286	7
		386	6
		486	2

Encoding

100000*sw* *mod*,001, *r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *reg,immed*	`or dx,110110b`	88/86	4
		286	3
		386	2
		486	1
OR *mem,immed*	`or flag_rec,8`	88/86	(*b*=17,*w*=25)+*EA*
		286	7
		386	7
		486	3

Encoding

0000110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *accum,immed*	`or ax,40h`	88/86	4
		286	3
		386	2
		486	1

OUT Output to Port

Transfers a byte or word (or a doubleword on the 80386–80486) to a port from the accumulator register. The port address is specified by the destination operand, which can be DX or an 8-bit constant. In protected mode, a general-protection fault occurs if **OUT** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 1110011*w* *data (1)*

Syntax	Examples	CPU	Clock Cycles
OUT *immed8,accum*	`out 60h,al`	88/86	10 (*88*=14)
		286	3
		386	10,*pm*=4,24*
		486	16,*pm*=11,31*

Encoding 1110111*w*

Syntax	Examples	CPU	Clock Cycles
OUT DX,*accum*	`out dx,ax`	88/86	8 (*88*=12)
	`out dx,al`	286	3
		386	11,*pm*=5,25*
		486	16,*pm*=10,30*

* First protected-mode timing: CPL < IOPL. Second timing: CPL > IOPL.

OUTS/OUTSB/OUTSW/OUTSD Output String to Port

80186–80486 Only Sends a string to a port. The string is considered the source and must be pointed to by DS:SI (even if an operand is given). The output port is specified in DX. For each element sent, SI is adjusted according to the size of the operand and the status of the direction flag. SI is increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **OUTS** form of the instruction is used, an operand must be provided to indicate the size of data elements to be sent. A segment override can be given. If **OUTSB** (bytes), **OUTSW** (words), or **OUTSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be sent.

OUTS and its variations are normally used with the **REP** prefix. Before the instruction is executed, CX should contain the number of elements to send. In protected mode, a general-protection fault occurs if **OUTS** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 0110111*w*

Syntax	Examples	CPU	Clock Cycles
OUTS DX, ⟦*segreg*:⟧ *src*	`rep   outs`	88/86	—
OUTSB ⟦**DX,** ⟦*segreg*:⟧ *src*⟧	`dx,buffer`	286	5
OUTSW ⟦**DX,** ⟦*segreg*:⟧ *src*⟧	`outsb`	386	14,*pm*=8,28*
OUTSD ⟦**DX,** ⟦*segreg*:⟧ *src*⟧	`rep   outsw`	486	17,*pm*=10,32*

* First protected-mode timing: CPL < IOPL. Second timing: CPL > IOPL.

POP Pop

Pops the top of the stack into the destination operand. The value at SS:SP is copied to the destination operand and SP is increased by 2. The destination operand can be a memory location, a general-purpose 16-bit register, or any segment register except CS. Use **RET** to pop CS. On the 80386–80486, 32-bit values can be popped by giving a 32-bit operand. ESP is increased by 4 for 32-bit pops.

Flags No change

Encoding 01011 *reg*

Syntax	Examples	CPU	Clock Cycles
POP *reg16*	`pop  cx`	88/86	8 (*88*=12)
POP *reg32**		286	5
		386	4
		486	1

Encoding 10001111 *mod*,000,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
POP *mem16*	`pop  param`	88/86	17+*EA* (*88*=25+*EA*)
POP *mem32**		286	5
		386	5
		486	6

Encoding 000,*sreg*,111

Syntax	Examples	CPU	Clock Cycles
POP *segreg*	`pop es`	88/86	8 (*88*=12)
	`pop ds`	286	5,*pm*=20
	`pop ss`	386	7,*pm*=21
		486	3,*pm*=9

Encoding 00001111 10,*sreg*,001

Syntax	Examples	CPU	Clock Cycles
POP *segreg**	`pop fs`	88/86	—
	`pop gs`	286	—
		386	7,*pm*=21
		486	3,*pm*=9

* 80386–80486 only.

POPA/POPAD Pop All

80186-80486 Only Pops the top 16 bytes on the stack into the eight general-purpose registers. The registers are popped in the following order: DI, SI, BP, SP, BX, DX, CX, AX. The value for the SP register is actually discarded rather than copied to SP. **POPA** always pops into 16-bit registers. On the 80386–80486, use **POPAD** to pop into 32-bit registers.

Flags No change

Encoding 01100001

Syntax	Examples	CPU	Clock Cycles
POPA	`popa`	88/86	—
POPAD*		286	19
		386	24
		486	9

* 80386–80486 only.

POPF/POPFD Pop Flags

Pops the value on the top of the stack into the flags register. **POPF** always pops into the 16-bit flags register. On the 80386–80486, use **POPFD** to pop into the 32-bit flags register.

Flags

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

Encoding

10011101

Syntax	Examples	CPU	Clock Cycles
POPF	`popf`	88/86	8 (*88*=12)
POPFD*		286	5
		386	5
		486	9,*pm*=6

* 80386–80486 only.

PUSH/PUSHW/PUSHD Push

Pushes the source operand onto the stack. SP is decreased by 2 and the source value is copied to SS:SP. The operand can be a memory location, a general-purpose 16-bit register, or a segment register. On the 80186–80486 processors, the operand can also be a constant. On the 80386–80486, 32-bit values can be pushed by specifying a 32-bit operand. ESP is decreased by 4 for 32-bit pushes. On the 8088 and 8086, **PUSH SP** saves the value of SP after the push. On the 80186–80486 processors, **PUSH SP** saves the value of SP before the push. The **PUSHW** and **PUSHD** instructions push a word (2 bytes) and a doubleword (4 bytes), respectively.

Flags

No change

Encoding

01010 *reg*

Syntax	Examples	CPU	Clock Cycles
PUSH *reg16*	`push dx`	88/86	11 (*88*=15)
PUSH *reg32**		286	3
PUSHW *reg16*		386	2
PUSHD *reg32**		486	1

Encoding 11111111 *mod*, 110,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
PUSH *mem16*	`push [di]`	88/86	16+*EA* (*88*=24+*EA*)
PUSH *mem32**	`push fcount`	286	5
		386	5
		486	4

Encoding 00,*sreg*,110

Syntax	Examples	CPU	Clock Cycles
PUSH *segreg*	`push es`	88/86	10 (*88*=14)
PUSHW *segreg*	`push ss`	286	3
PUSHD *segreg**	`push cs`	386	2
		486	3

Encoding 00001111 10,*sreg*,000

Syntax	Examples	CPU	Clock Cycles
PUSH *segreg*	`push fs`	88/86	—
PUSHW *segreg*	`push gs`	286	—
PUSHD *segreg**		386	2
		486	3

Encoding 011010*s*0 *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
PUSH *immed*	`push 'a'`	88/86	—
PUSHW *immed*	`push 15000`	286	3
PUSHD *immed**		386	2
		486	1

* 80386–80486 only.

PUSHA/PUSHAD Push All

80186–80486 Only Pushes the eight general-purpose registers onto the stack. The registers are pushed in the following order: AX, CX, DX, BX, SP, BP, SI, DI. The value pushed for SP is the value before the instruction. **PUSHA** always pushes 16-bit registers. On the 80386–80486, use **PUSHAD** to push 32-bit registers.

Flags No change

Encoding 01100000

Syntax	Examples	CPU	Clock Cycles
PUSHA	`pusha`	88/86	—
PUSHAD*		286	17
		386	18
		486	11

* 80386–80486 only.

PUSHF/PUSHFD Push Flags

Pushes the flags register onto the stack. **PUSHF** always pushes the 16-bit flags register. On the 80386–80486, use **PUSHFD** to push the 32-bit flags register.

Flags No change

Encoding 10011100

Syntax	Examples	CPU	Clock Cycles
PUSHF	`pushf`	88/86	10(*88*=14)
PUSHFD*		286	3
		386	4
		486	4,*pm*=3

* 80386–80486 only.

RCL/RCR/ROL/ROR Rotate

Rotates the bits in the destination operand the number of times specified in the source operand. **RCL** and **ROL** rotate the bits left; **RCR** and **ROR** rotate right.

ROL and **ROR** rotate the number of bits in the operand. For each rotation, the leftmost or rightmost bit is copied to the carry flag as well as rotated. **RCL** and **RCR** rotate through the carry flag. The carry flag becomes an extension of the operand so that a 9-bit rotation is done for 8-bit operands, or a 17-bit rotation for 16-bit operands.

On the 8088 and 8086, the source operand can be either CL or 1. On the 80186–80486, the source operand can be CL or an 8-bit constant. On the 80186–80486, rotate counts larger than 31 are masked off, but on the 8088 and 8086, larger rotate counts are performed despite the inefficiency involved. The overflow flag is modified only by single-bit variations of the instruction; for multiple-bit variations, the overflow flag is undefined.

Flags

O	D	I	T	S	Z	A	P	C
±								±

Encoding

1101000*w* *mod, TTT*,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,**1**	`ror ax,1`	88/86	2
ROR *reg*,**1**	`rol dl,1`	286	2
		386	3
		486	3
RCL *reg*,**1**	`rcl dx,1`	88/86	2
RCR *reg*,**1**	`rcr bl,1`	286	2
		386	9
		486	3
ROL *mem*,**1**	`ror bits,1`	88/86	15+*EA* (*W88*=23+*EA*)
ROR *mem*,**1**	`rol WORD PTR [bx],1`	286	7
		386	7
		486	4
RCL *mem*,**1**	`rcl WORD PTR [si],1`	88/86	15+*EA* (*W88*=23+*EA*
RCR *mem*,**1**	`rcr WORD PTR m32[0],1`	286	7
		386	10
		486	4

Encoding

1101001*w* *mod, TTT*,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,**CL**	`ror ax,cl`	88/86	8+4*n*
ROR *reg*,**CL**	`rol dx,cl`	286	5+*n*
		386	3
		486	3
RCL *reg*,**CL**	`rcl dx,cl`	88/86	8+4*n*
RCR *reg*,**CL**	`rcr bl,cl`	286	5+*n*
		386	9
		486	8–30
ROL *mem*,**CL**	`ror color,cl`	88/86	20+*EA*+4*n*
ROR *mem*,**CL**			(*W88*=28+*EA*+4*n*)
	`rol WORD PTR [bp+6],cl`	286	8+*n*
		386	7
		486	4

Syntax	Examples	CPU	Clock Cycles
RCL *mem*,**CL**	`rcr WORD PTR [bx+di],cl`	88/86	20+*EA*+4*n*
RCR *mem*,**CL**			(*W88*=28+*EA*+4*n*)
	`rcl masker`	286	8+*n*
		386	10
		486	9–31

Encoding 1100000*w* *mod,TTT*,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg,immed8*	`rol ax,13`	88/86	—
ROR *reg,immed8*	`ror bl,3 286`	286	5+*n*
		386	3
		486	2
RCL *reg,immed8*	`rcl bx,5`	88/86	—
RCR *reg,immed8*	`rcr si,9`	286	5+*n*
		386	9
		486	8–30
ROL *mem,immed8*	`rol BYTE PTR [bx],10`	88/86	—
ROR *mem,immed8*	`ror bits,6`	286	8+*n*
		386	7
		486	4
RCL *mem,immed8*	`rcl WORD PTR [bp+8],`	88/86	—
RCR *mem,immed8*	`rcr masker,3`	286	8+*n*
		386	10
		486	9–31

* *TTT* represents one of the following bit codes: 000 for **ROL**, 001 for **ROR**, 010 for **RCL**, or 011 for **RCR**.

REP Repeat String

Repeats a string instruction the number of times indicated by CX. First, CX is compared to 0; if it equals 0, execution proceeds to the next instruction. Otherwise, CX is decremented, the string instruction is performed, and the loop continues. **REP** is used with **MOVS** and **STOS**. **REP** also can be used with **INS** and **OUTS** on the 80186–80486 processors. On all processors except the 80386–80486, combining a repeat prefix with a segment override can cause errors if an interrupt occurs.

Flags No change

Encoding 11110011 1010010*w*

Syntax	Examples	CPU	Clock Cycles
REP MOVS *dest*,*src*	`rep movs source,dest`	88/86	9+17*n* (*W88*=9+25*n*)
REP MOVSB ⟦*dest*,*src*⟧	`rep movsw`	286	5+4*n*
REP MOVSW ⟦*dest*,*src*⟧		386	7+4*n*
REP MOVSD ⟦*dest*,*src*⟧*		486	12+3*n*#

Encoding 11110011 1010101*w*

Syntax	Examples	CPU	Clock Cycles
REP STOS *dest*	`rep stosb`	88/86	9+10*n* (*W88*=9+14*n*)
REP STOSB ⟦*dest*⟧	`rep stos dest`	286	4+3*n*
REP STOSW ⟦*dest*⟧		386	5+5*n*
REP STOSD ⟦*dest*⟧*		486	7+4*n*†

Encoding 11110011 1010101*w*

Syntax	Examples	CPU	Clock Cycles
REP LODS *dest*	`rep lodsb`	88/86	—
REP LODSB ⟦*dest*⟧	`rep lods dest`	286	—
REP LODSW ⟦*dest*⟧		386	—
REP LODSD ⟦*dest*⟧*		486	7+4*n*†

Encoding 11110011 0110110*w*

Syntax	Examples	CPU	Clock Cycles
REP INS *dest*,**DX**	`rep insb`	88/86	—
REP INSB ⟦*dest*,**DX**⟧	`rep ins dest,dx`	286	5+4*n*
REP INSW ⟦*dest*,**DX**⟧		386	13+6*n*,*pm*=(7,27)+6*n*§
REP INSD ⟦*dest*,**DX**⟧*			16+8*n*,*pm*=(10,30)+8*n*
		486	§

Encoding 11110011 0110111*w*

Syntax	Examples	CPU	Clock Cycles
REP OUTS DX,*src*	`rep outs dx,source`	88/86	—
REP OUTSB ⟦*src*⟧	`rep outsw`	286	5+4*n*
REP OUTSW ⟦*src*⟧		386	12+5*n*,*pm*=(6,26)+5*n*§
REP OUTSD ⟦*src*⟧*		486	17+5*n*,*pm*=(11,31)+5*n*§

* 80386–80486 only.

# 5 if *n* = 0, 13 if *n* = 1.

† 5 if *n* = 0.

§ First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

REP*condition* Repeat String Conditionally

Repeats a string instruction as long as *condition* is true and the maximum count has not been reached. **REPE** and **REPZ** (they are synonyms) repeat while the zero flag is set. **REPNE** and **REPNZ** (they are synonyms) repeat while the zero flag is cleared. The conditional-repeat prefixes should only be used with **SCAS** and **CMPS**, since these are the only string instructions that modify the zero flag. Before executing the instruction, CX should be set to the maximum allowable number of repetitions. First, CX is compared to 0; if it equals 0, execution proceeds to the next instruction. Otherwise, CX is decremented, the string instruction is performed, and the loop continues. On all processors except the 80386–80486, combining a repeat prefix with a segment override may cause errors if an interrupt occurs during a string operation.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding

11110011 1010011*w*

Syntax	Examples	CPU	Clock Cycles
REPE CMPS *src*,*dest*	`repz cmpsb`	88/86	9+22*n* (*W88*=9+30*n*)
REPE CMPSB ⟦*src*,*dest*⟧	`repe cmps`	286	5+9*n*
REPE CMPSW ⟦*src*,*dest*⟧	`src,dest`	386	5+9*n*
REPE CMPSD ⟦*src*,*dest*⟧*		486	7+7*n*#

Encoding

11110011 1010111*w*

Syntax	Examples	CPU	Clock Cycles
REPE SCAS *dest*	`repe scas dest`	88/86	9+15*n* (*W88*=9+19*n*)
REPE SCASB ⟦*dest*⟧	`repz scasw`	286	5+8*n*
REPE SCASW ⟦*dest*⟧		386	5+8*n*
REPE SCASD ⟦*dest*⟧*		486	7+5*n*#

Encoding

11110010 1010011*w*

Syntax	Examples	CPU	Clock Cycles
REPNE CMPS *src*,*dest*	`repne cmpsw`	88/86	9+22*n* (*W88*=9+30*n*)
REPNE CMPSB ⟦*src*,*dest*⟧	`repnz cmps`	286	5+9*n*
REPNE CMPSW ⟦*src*,*dest*⟧	`src,dest`	386	5+9*n*
REPNE CMPSD ⟦*src*,*dest*⟧*		486	7+7*n*#

Encoding 11110010 1010111*w*

Syntax	Examples	CPU	Clock Cycles
REPNE SCAS *des*	`repne scas dest`	88/86	9+15*n* (*W88*=9+19*n*)
REPNE SCASB ⟦*dest*⟧	`repnz scasb`	286	5+8*n*
REPNE SCASW ⟦*dest*⟧		386	5+8*n*
REPNE SCASD ⟦*dest*⟧*		486	7+5*n**

* 80386–80486 only.

# 5 if n=0.

RET/RETN/RETF Return from Procedure

Returns from a procedure by transferring control to an address popped from the top of the stack. A constant operand can be given indicating the number of additional bytes to release. The constant is normally used to adjust the stack for arguments pushed before the procedure was called. The size of a return (near or far) is the size of the procedure in which the **RET** is defined with the **PROC** directive. **RETN** can be used to specify a near return; **RETF** can specify a far return. A near return pops a word into IP. A far return pops a word into IP and then pops a word into CS. After the return, the number of bytes given in the operand (if any) is added to SP.

Flags No change

Encoding 11000011

Syntax	Examples	CPU	Clock Cycles
RET	`ret`	88/86	16 (*88*=20)
RETN	`retn`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11000010 *data (2)*

Syntax	Examples	CPU	Clock Cycles
RET *immed16*	`ret 2`	88/86	20 (*88*=24)
RETN *immed16*	`retn 8`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11001011

Syntax	Examples	CPU	Clock Cycles
RET	`ret`	88/86	26 (*88*=34)
RETF	`retf`	286	15+*m*,*pm*=25+*m*,55*
		386	18+*m*,*pm*=32+*m*,62*
		486	13,*pm*=18,33*

Encoding 11001010 *data (2)*

Syntax	Examples	CPU	Clock Cycles
RET *immed16*	`ret 8`	88/86	25 (*88*=33)
RETF *immed16*	`retf 32`	286	15+*m*,*pm*=25+*m*,55*
		386	18+*m*,*pm*=32+*m*,68*
		486	14,*pm*=17,33*

* The first protected-mode timing is for a return to the same privilege level; the second is for a return to a lesser privilege level.

ROL/ROR Rotate

See **RCL/RCR**.

SAHF Store AH into Flags

Transfers AH into bits 0 to 7 of the flags register. This includes the carry, parity, auxiliary carry, zero, and sign flags, but not the trap, interrupt, direction, or overflow flags.

Flags

O	D	I	T	S	Z	A	P	C
				±	±	±	±	±

Encoding 10011110

Syntax	Examples	CPU	Clock Cycles
SAHF	`sahf`	88/86	4
		286	2
		386	3
		486	2

SAL/SAR Shift

See **SHL/SHR/SAL/SAR**.

SBB Subtract with Borrow

Adds the carry flag to the second operand, then subtracts that value from the first operand. The result is assigned to the first operand. **SBB** is used to subtract the least significant portions of numbers that must be processed in multiple registers.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 000110*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *reg,reg*	`sbb dx,cx`	88/86	3
		286	2
		386	2
		486	1
SBB *mem,reg*	`sbb WORD PTR m32[2],dx`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3
SBB *reg,mem*	`sbb dx,WORD PTR m32[2]`	88/86	9+*EA* (*W88*=13+*EA*)
		286	7
		386	7
		486	2

Encoding 100000*sw* *mod,011, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *reg,immed*	`sbb dx,45`	88/86	4
		286	3
		386	2
		486	1
SBB *mem,immed*	`sbb WORD PTR m32[2],40`	88/86	17+*EA* (*W88*=25+*EA*)
		286	7
		386	7
		486	3

Encoding 0001110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *accum,immed*	`sbb ax,320 88/86`	4	
		86	3
		386	2
		486	1

SCAS/SCASB/SCASW/SCASD Scan String Flags

Scans a string to find a value specified in the accumulator register. The string to be scanned is considered the destination. ES:DI must point to that string, even if an operand is specified. For each element, the destination element is subtracted from the accumulator value and the flags are updated to reflect the result (although the result is not stored). DI is adjusted according to the size of the operands and the status of the direction flag. DI is increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **SCAS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. No segment override is allowed. If **SCASB** (bytes), **SCASW** (words), or **SCASD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element scanned for is in AL, AX, or EAX.

SCAS and its variations are normally used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first element in a string that matches the value in the accumulator register. **REPE** (or **REPZ**) is used to find the first mismatch. Before the scan, CX should contain the maximum number of elements to scan. After a **REPNE SCAS**, the zero flag is clear if the string does not contain the accumulator value. After a **REPE SCAS**, the zero flag is set if the string contains nothing but the accumulator value.

When the instruction finishes, ES:DI points to the element that follows (if the direction flag is clear) or precedes (if the direction flag is set) the match or mismatch. If CX decrements to 0, ES:DI points to the element that follows or precedes the last comparison. The zero flag is set or clear according to the result of the last comparison, not according to the value of CX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1010111*w*

Syntax	Examples	CPU	Clock Cycles
SCAS ⟦**ES:**⟧ *dest*	`repne scasw`	88/86	15 (*W88*=19)
SCASB ⟦⟦**ES:**⟧ *dest*⟧	`repe scasb`	286	7
SCASW ⟦⟦**ES:**⟧ *dest*⟧	`scas es:destin`	386	7
SCASD ⟦⟦**ES:**⟧ *dest*⟧*		486	6

* 80386–80486 only

SET*condition* Set Conditionally

80386–80486 Only Sets the byte specified in the operand to 1 if *condition* is true or to 0 if *condition* is false. The condition is tested by checking the flags shown in the table on the following page. The instruction is used to set Boolean flags conditionally.

Flags No change

Encoding 00001111 1001*cond* *mod*,000,*r/m*

Syntax	Examples	CPU	Clock Cycles
SET*condition reg8*	`setc dh`	88/86	—
	`setz al`	286	—
	`setae bl`	386	4
		486	true=4, false=3
SET*condition mem8*	`seto BTYE PTR [ebx]`	88/86	—
	`setle flag`	286	—
	`sete Booleans[di]`	386	5
		486	true=3, false=4

Set Conditions

Opcode	Mnemonic	Flags Checked	Description
10010010	**SETB/SETNAE**	CF=1	Set if below/not above or equal (unsigned comparisons)
10010011	**SETAE/SETNB**	CF=0	Set if above or equal/not below (unsigned comparisons)
10010110	**SETBE/SETNA**	CF=1 or ZF=1	Set if below or equal/not above (unsigned comparisons)
10010111	**SETA/SETNBE**	CF=0 and ZF=0	Set if above/not below or equal (unsigned comparisons)
10010100	**SETE/SETZ**	ZF=1	Set if equal/zero
10010101	**SETNE/SETNZ**	ZF=0	Set if not equal/not zero
Opcode	**Mnemonic**	**Flags Checked**	**Description**

10011100	**SETL/SETNGE**	SF_OF	Set if less/not greater or equal (signed comparisons)
10011101	**SETGE/SETNL**	SF=OF	Set if greater or equal/not less (signed comparisons)
10011110	**SETLE/SETNG**	ZF=1 or SF_OF	Set if less or equal/not greater or equal (signed comparisons)
10011111	**SETG/SETNLE**	ZF=0 and SF=OF	Set if greater/not less or equal (signed comparisons)
10011000	**SETS**	SF=1	Set if sign
10011001	**SETNS**	SF=0	Set if not sign
10010010	**SETC**	F=1	Set if carry
10010011	**SETNC**	CF=0	Set if not carry
10010000	**SETO**	OF=1	Set if overflow
10010001	**SETNO**	OF=0	Set if not overflow
10011010	**SETP/SETPE**	PF=1	Set if parity/parity even
10011011	**SETNP/SETPO**	PF=0	Set if no parity/parity odd

SGDT/SIDT/SLDT Store Descriptor Table

80286-80486 Only Stores a descriptor table register into a specified operand. **SGDT** stores the Global Descriptor Table; **SIDT**, the Interrupt Vector Table; and **SLDT**, the Local Descriptor Table. These instructions are generally useful only in privileged mode. See Intel documentation for details on descriptor tables and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod*,000,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
SGDT *mem48*	`sgdt descriptor`	88/86	—
		286	11
		386	9
		486	10

Encoding 00001111 00000001 *mod*,001,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
SIDT *mem48*	`sidt descriptor`	88/86	—
		286	12
		386	9
		486	10

Encoding 00001111 00000000 *mod,* 000,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SLDT *reg16*	`sldt ax`	88/86	—
		286	2
		386	2
		486	2
SLDT *mem16*	`sldt selector`	88/86	—
		286	3
		386	2
		486	3

SHL/SHR/SAL/SAR Shift

Shifts the bits in the destination operand the number of times specified by the source operand. **SAL** and **SHL** shift the bits left; **SAR** and **SHR** shift right.

With **SHL**, **SAL**, and **SHR**, the bit shifted off the end of the operand is copied into the carry flag, and the leftmost or rightmost bit opened by the shift is set to 0. With **SAR**, the bit shifted off the end of the operand is copied into the carry flag, and the leftmost bit opened by the shift retains its previous value (thus preserving the sign of the operand). **SAL** and **SHL** are synonyms.

On the 8088 and 8086, the source operand can be either CL or 1. On the 80186–80486 processors, the source operand can be CL or an 8-bit constant. On the 80186–80486 processors, shift counts larger than 31 are masked off, but on the 8088 and 8086, larger shift counts are performed despite the inefficiency. Only single-bit variations of the instruction modify the overflow flag; for multiple-bit variations, the overflow flag is undefined.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	?	±	±

Encoding 1101000*w* mod,*TTT**,r/m *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg*,**1**	`sar di,1`	88/86	2
	`sar cl,1`	286	2
		386	3
		486	3
SAL *reg*,**1**	`shr dh,1`	88/86	2
SHL *reg*,**1**	`shl si,1`	286	2
SHR *reg*,**1**	`sal bx,1`	386	3
SAR *mem*,**1**	`sar count,1`	486	3
		88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	7
		486	4
SAL *mem*,**1**	`sal WORD PTR m32[0],1`	88/86	15+*EA* (*W88*=23+*EA*)
SHL *mem*,**1**	`shl index,1`	286	7
SHR *mem*,**1**	`shr unsign[di],1`	386	7
		486	4

Encoding 1101001*w* *mod,TTT*,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg*,**CL**	`sar bx,cl`	88/86	8+4*n*
	`sar dx,cl`	286	5+*n*
		386	3
		486	3
SAL *reg*,**CL**	`shr dx,cl`	88/86	8+4*n*
SHL *reg*,**CL**	`shl di,cl`	286	5+*n*
SHR *reg*,**CL**	`sal ah,cl`	386	3
		486	3
SAR *mem*,**CL**	`sar sign,cl`	88/86	20+*EA*+4*n* (*W88*=28+*EA*+4*n*)
	`sar WORD PTR [bp+8],cl`	286	8+*n*
		386	7
		486	4
SAL *mem*,**CL**	`shr WORD PTR m32[2],cl`	88/86	20+*EA*+4*n* (*W88*=28+*EA*+4*n*)
SHL *mem*,**CL**	`sal BYTE PTR [di],cl`		
SHR *mem*,**CL**	`shl index,cl`	286	8+*n*
		386	7
		486	4

Encoding 1100000w *mod,TTT*,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles

SAR *reg,immed8*	`sar bx,5`
	`sar cl,5`
		88/86
		286
		386
		486
			—
			5+*n*
			3
			2

SAL *reg,immed8*
SHL *reg,immed8*
SHR *reg,immed8*
	`sal cx,6`
	`shl di,2`
	`shr bx,8`
		88/86
		286
		386
		486
			—
			5+*n*
			3
			2

SAR *mem,immed8*	`sar sign_count,3`
	`sar WORD PTR [bx],5`
		88/86
		286
		386
		486
			—
			8+*n*
			7
			4

SAL *reg,immed8*
SHL *reg,immed8*
SHR *reg,immed8*
	`shr mem16,11`
	`shl unsign,4`
	`sal array[bx+di],14`
		88/86
		286
		386
		486
			—
			8+*n*
			7
			4

* *TTT* represents one of the following bit codes: 100 for **SHL** or **SAL**, 101 for **SHR**, or 111 for **SAR**.

SHLD/SHRD Double Precision Shift

80386–80486 Only Shifts the bits of the second operand into the first operand. The number of bits shifted is specified by the third operand. **SHLD** shifts the first operand to the left by the number of positions specified in the count. The positions opened by the shift are filled by the most significant bits of the second operand. **SHRD** shifts the first operand to the right by the number of positions specified in the count. The positions opened by the shift are filled by the least significant bits of the second operand. The count operand can be either CL or an 8-bit constant. If a shift count larger than 31 is given, it is adjusted by using the remainder (modulo) of a division by 32.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	±

Encoding 00001111 10100100 *mod,reg,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SHLD *reg16,reg16,immed8*	`shld ax,dx,10`	88/86	—
SHLD *reg32,reg32,immed8*		286	—
		386	3
		486	2
SHLD *mem16,reg16,immed8*	`shld bits,cx,5`	88/86	—
SHLD *mem32,reg32,immed8*		286	—
		386	7
		486	3

Encoding 00001111 10101100 *mod,reg,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SHRD *reg16,reg16,immed8*	`shrd cx,si,3`	88/86	—
SHRD *reg32,reg32,immed8*		286	—
		386	3
		486	2
SHRD *mem16,reg16,immed8*	`shrd [di],dx,13`	88/86	—
SHRD *mem32,reg32,immed8*		286	—
		386	7
		486	3

Encoding 00001111 10100101 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SHLD *reg16,reg16,***CL**	`shld ax,dx,cl`	88/86	—
SHLD *reg32,reg32,***CL**		286	—
		386	3
		486	3
SHLD *mem16,reg16,***CL**	`shld`	88/86	—
SHLD *mem32,reg32,***CL**	`masker,ax,cl`	286	—
		386	7
		486	4

Encoding 00001111 10101101 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SHRD *reg16,reg16*,**CL**	`shrd bx,dx,cl`	88/86	—
SHRD *reg32,reg32*,**CL**		286	—
		386	3
		486	3
SHRD *mem16,reg16*,**CL**	`shrd [bx],dx,cl`	88/86	—
SHRD *mem32,reg32*,**CL**		286	—
		386	7
		486	4

SMSW Store Machine Status Word

80286-80486 Only Stores the Machine Status Word (MSW) into a specified memory operand. **SMSW** is generally useful only in protected mode. See Intel documentation for details on the MSW and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod*,100,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SMSW *reg16*	`smsw ax`	88/86	—
		286	2
		386	2
		486	2
SMSW *mem16*	`smsw machine`	88/86	—
		286	3
		386	3
		486	3

STC Set Carry Flag

Sets the carry flag.

Flags

```
O D I T S Z A P C
                1
```

Encoding 11111001

Syntax	Examples	CPU	Clock Cycles
STC	`stc`	88/86	2
		286	2
		386	2
		486	2

STD Set Direction Flag

Sets the direction flag. All subsequent string instructions will process down (from high addresses to low addresses).

Flags

```
O D I T S Z A P C
  1
```

Encoding 11111101

Syntax	Examples	CPU	Clock Cycles
STD	`std`	88/86	2
		286	2
		386	2
		486	2

STI Set Interrupt Flag

Sets the interrupt flag. When the interrupt flag is set, maskable interrupts are recognized. If interrupts were disabled by a previous **CLI** instruction, pending interrupts will not be executed immediately; they will be executed after the instruction following **STI**.

Flags

```
O D I T S Z A P C
    1
```

Encoding 11111011

Syntax	Examples	CPU	Clock Cycles
STI	`sti`	88/86	2
		286	2
		386	3
		486	5

STOS/STOSB/STOSW/STOSD Store String Data

Stores the value of the accumulator in a string. The string is the destination and must be pointed to by ES:DI, even if an operand is given. For each source element loaded, DI is adjusted according to the size of the operand and the status of the direction flag. DI is incremented if the direction flag has been cleared with **CLD** or decremented if the direction flag has been set with **STD**.

If the **STOS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. No segment override is allowed. If **STOSB** (bytes), **STOSW** (words), or **STOSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element comes from AL, AX, or EAX.

STOS and its variations are often used with the **REP** prefix to fill a string with a repeated value. Before the repeated instruction is executed, CX should contain the number of elements to store.

Flags No change

Encoding 1010101*w*

Syntax	Examples	CPU	Clock Cycles
STOS ⟦**ES:**⟧ *dest*	`stos es:dstring`	88/86	11 (*W88*=15)
STOSB ⟦⟦**ES:**⟧ *dest*⟧	`rep  stosw`	286	3
STOSW ⟦⟦**ES:**⟧ *dest*⟧	`rep  stosb`	386	4
STOSD ⟦⟦**ES:**⟧ *dest*⟧*		486	5

* 80386–80486 only

STR Store Task Register

80286-80486 Only Stores the current task register to the specified operand. This instruction is generally useful only in privileged mode. See Intel documentation for details on task registers and other protected-mode concepts.

Flags No change

Encoding 00001111 00000000 *mod*, 001, *reg* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
STR *reg16*	`str cx`	88/86	—
		286	2
		386	2
		486	2
STR *mem16*	`str taskreg`	88/86	—
		286	3
		386	2
		486	3

SUB Subtract

Subtracts the source operand from the destination operand and stores the result in the destination operand.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 001010*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *reg,reg*	`sub ax,bx`	88/86	3
	`sub bh,dh`	286	2
		386	2
		486	1
SUB *mem,reg*	`sub tally,bx`	88/86	16+*EA* (*W88*=24+*EA*)
	`sub array[di],bl`	286	7
		386	6
		486	3

Syntax	Examples	CPU	Clock Cycles
SUB *reg,mem*	`sub cx,discard`	88/86	9+*EA* (*W88*=13+*EA*)
	`sub al,[bx]`	286	7
		386	7
		486	2

Encoding 100000*sw* *mod*,101,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *reg,immed*	`sub dx,45`	88/86	4
	`sub bl,7`	286	3
		386	2
		486	1
SUB *mem,immed*	`sub total,4000`	88/86	17+*EA* (*W88*=25+*EA*)
	`sub BYTE PTR [bx+di],2`	286	7
		386	7
		486	3

Encoding 0010110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *accum,immed*	`sub ax,32000`	88/86	4
		286	3
		386	2
		486	1

TEST Logical Compare

Tests specified bits of an operand and sets the flags for a subsequent conditional jump or set instruction. One of the operands contains the value to be tested. The other contains a bit mask indicating the bits to be tested. **TEST** works by doing a bitwise AND operation on the source and destination operands. The flags are modified according to the result, but the destination operand is not changed. This instruction is the same as the **AND** instruction, except the result is not stored.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 1000010*w* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *reg,reg*	`test dx,bx`	88/86	3
	`test bl,ch`	286	2
		386	2
		486	1
TEST *mem,reg*	`test dx,flags`	88/86	9+*EA* (*W88*=13+*EA*)
TEST *reg,mem**	`test bl,bitarray[bx]`	286	6
		386	5
		486	2

Encoding 1111011*w* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *reg,immed*	`test cx,30h`	88/86	5
	`test cl,1011b`	286	3
		386	2
		486	1
TEST *mem,immed*	`test masker,1`	88/86	11+*EA*
	`test BYTE PTR [bx],03h`	286	6
		386	5
		486	2

Encoding 1010100*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *accum,immed*	`test ax,90h`	88/86	4
		286	3
		386	2
		486	1

* MASM transposes **TEST** *reg***,***mem*; that is, it is encoded as **TEST** *mem***,***reg*.

VERR/VERW Verify Read or Write

80286-80486 Protected Only Verifies that a specified segment selector is valid and can be read or written to at the current privilege level. **VERR** verifies that the selector is readable. **VERW** verifies that the selector can be written to. If the segment is verified, the zero flag is set. Otherwise, the zero flag is cleared.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 00000000 *mod*, 100,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
VERR *reg16*	`verr ax`	88/86	—
		286	14
		386	10
		486	11
VERR *mem16*	`verr selector`	88/86	—
		286	16
		386	11
		486	11

Encoding 00001111 00000000 *mod*, 101,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
VERW *reg16*	`verw cx`	88/86	—
		286	14
		386	15
		486	11
VERW *mem16*	`verw selector`	88/86	—
		286	16
		386	16
		486	11

WAIT Wait

Suspends processor execution until the processor receives a signal that a coprocessor has finished a simultaneous operation. It should be used to prevent a coprocessor instruction from modifying a memory location that is being modified simultaneously by a processor instruction. **WAIT** is the same as the coprocessor **FWAIT** instruction.

Flags No change

Encoding 10011011

Syntax	Examples	CPU	Clock Cycles
WAIT	`wait`	88/86	4
		286	3
		386	6
		486	1–3

WBINVD Write Back and Invalidate Data Cache

80486 Only Empties the contents of the current data cache after writing changes to memory. Proper use of this instruction requires knowledge of how contents are placed in the cache. **WBINVD** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00001001

Syntax	Examples	CPU	Clock Cycles
WBINVD	`wbinvd`	88/86	—
		286	—
		386	—
		486	5

XADD Exchange and Add

80486 Only Adds the source and destination operands and stores the sum in the destination; simultaneously, the original value of the destination is moved to the source. The instruction sets flags according to the result of the addition.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 00001111 1100000*b* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XADD *mem,reg*	`xadd warr[bx],ax`	88/86	—
	`xadd string,bl`	286	—
		386	—
		486	4
XADD *reg,reg*	`xadd dl,al`	88/86	—
	`xadd bx,dx`	286	—
		386	—
		486	3

XCHG Exchange

Exchanges the values of the source and destination operands.

Flags No change

Encoding 1000011*w* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XCHG *reg,reg*	`xchg cx,dx`	88/86	4
	`xchg bl,dh`	286	3
	`xchg al,ah`	386	3
		486	3
XCHG *reg,mem*	`xchg [bx],ax`	88/86	17+*EA* (*W88*=25+*EA*)
XCHG *mem,reg*	`xchg bx,pointer`	286	5
		386	5
		486	5

Encoding 10010 *reg*

Syntax	Examples	CPU	Clock Cycles
XCHG *accum,reg16**	`xchg ax,cx`	88/86	3
XCHG *reg16,accum**	`xchg cx,ax`	286	3
		386	3
		486	3

* On the 80386–80486, the accumulator may also be exchanged with a 32-bit register.

XLAT/XLATB Translate

Translates a value from one coding system to another by looking up the value to be translated in a table stored in memory. Before the instruction is executed, BX should point to a table in memory and AL should contain the unsigned position of the value to be translated from the table. After the instruction, AL contains the table value at the specified position. No operand is required, but one can be given to specify a segment override. DS is assumed unless a segment override is given. **XLATB** is a synonym for **XLAT**. Either version allows an operand, but neither requires one.

Flags No change

Encoding 11010111

Syntax	Examples	CPU	Clock Cycles
XLAT ⟦⟦*segreg*:⟧ *mem*⟧	`xlat`	88/86	11
XLATB ⟦⟦*segreg*:⟧ *mem*⟧	`xlatb es:table`	286	5
		386	5
		486	4

XOR Exclusive OR

Performs a bitwise exclusive OR operation on the source and destination operands and stores the result in the destination. For each bit position in the operands, if both bits are set or if both bits are cleared, the corresponding bit of the result is cleared. Otherwise, the corresponding bit of the result is set.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 001100*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *reg,reg*	`xor cx,bx`	88/86	3
	`xor ah,al`	286	2
		386	2
		486	1
XOR *mem,reg*	`xor [bp+10],cx`	88/86	16+*EA* (*W88*=24+*EA*)
	`xor masked,bx`	286	7
		386	6
		486	3
XOR *reg,mem*	`xor cx,flags`	88/86	9+*EA* (*W88*=13+*EA*)
	`xor bl,bitarray[di]`	286	7
		386	7
		486	2

Encoding 100000*sw* *mod*,110,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *reg,immed*	`xor bx,10h`	88/86	4
	`xor bl,1`	286	3
		386	2
		486	1
XOR *mem,immed*	`xor Boolean,1`	88/86	17+*EA* (*W88*=25+*EA*)
	`xor switches[bx],101b`	286	7
		386	7
		486	3

Encoding 0011010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *accum,immed*	`xor ax,01010101b`	88/86	4
		286	3
		386	2
		486	1