Quick Reference

VERSION
4.0

Borland® Turbo Assembler®

Turbo Assembler®

Version 4.0

Quick Reference Guide

Borland International, Inc.
100 Borland Way, Scotts Valley, CA 95066-3249

PRINTED IN THE UNITED STATES OF AMERICA

2E0R0394
9495969798-9 8 7 6 5 4 3
H1

TABLE OF CONTENTS

P A R T 1

Predefined symbols

All the predefined symbols can be used in both MASM and Ideal mode.

$

Represents the current location counter within the current segment.

@32Bit

Numeric equate indicating whether segments in the current model are declared as 16 bit or 32 bit.

@code

Alias equate for .CODE segment name.

@CodeSize

Numeric equate that indicates code memory model (0=near, 1=far).

@CPU

Numeric equate that returns information about current processor directive.

@curseg

Alias equate for current segment.

@data

Alias equate for near data group name.

@DataSize

Numeric equate that indicates the data memory model (0=near, 1=far, 2=huge).

??date

String equate for today's date.

@fardata

Alias equate for initialized far data segment name.

@fardata?

Alias equate for uninitialized far data segment name.

@FileName

Alias equate for current assembly file name.

??filename

String equate for current assembly file name.

@Interface

Numeric equate indicating the language and operating system selected by MODEL.

@Model

Numeric equate representing the model currently in effect.

@Object

Text macro containing the name of the current object.
= Alias equate for stack segment.

@Stack

Alias equate for stack segment.

@Startup

Label that marks the beginning of startup code.

@Table_<*objectname*>

Data type containing the object's method table.

@TableAddr_<*objectname*>

Label describing the address of the instance of the object's virtual method table.

??time

String equate for the current time.

??version

Numeric equate for current Turbo Assembler version number.

@WordSize

Numeric equate that indicates 16- or 32-bit segments (2=16-bit, 4=32-bit).

P A R T 2

Operators

This part covers the operators Turbo Assembler provides and their precedence. The two tables that follow detail operator precedence for Ideal and MASM modes.

Ideal mode operator precedence

The following table lists the operators in order of priority (highest is first, lowest is last):

- (), [], **LENGTH, MASK, OFFSET, SEG, SIZE, WIDTH**
- **HIGH, LOW**
- +, - (unary)
- *, /, **MOD, SHL, SHR**
- +, - (binary)
- **EQ, GE, GT, LE, LT, NE**
- **NOT**
- **AND**
- **OR, XOR**
- : (segment override)
- . (structure member selector)
- **HIGH** (before pointer), **LARGE, LOW** (before pointer), **PTR, SHORT, SMALL, SYMTYPE**

MASM mode operator precedence

- <, (), [], **LENGTH, MASK, SIZE, WIDTH**
- . (structure member selector)
- **HIGH, LOW**
- +, - (unary)
- : (segment override)
- **OFFSET, PTR, SEG, THIS, TYPE**
- *, /, **MOD, SHL, SHR**
- +, - (binary)
- **EQ, GE, GT, LE, LT, NE**
- **NOT**
- **AND**
- **OR, XOR**
- **LARGE, SHORT, SMALL, .TYPE**

Operators

( ) — Ideal, MASM

(expression)

Marks *expression* for priority evaluation.

* — Ideal, MASM

*expression1 * expression2*

Multiplies two integer expressions. Also used with 80386 addressing modes where one expression is a register.

+ (binary) — Ideal, MASM

expression1 + expression2

Adds two expressions.

+ (unary) — Ideal, MASM

+ expression

Indicates that *expression* is positive.

- (binary) — Ideal, MASM

expression1 - expression2

Subtracts two expressions.

- (unary) — Ideal, MASM

- expression

Changes the sign of *expression*.

. — Ideal, MASM

memptr.fieldname

Selects a structure member.

/ — Ideal, MASM

expression1 / expression2

Divides two integer expressions.

:
Ideal, MASM

segorgroup : *expression*

Generates segment or group override.

?
Ideal, MASM

D*x* ?

Initializes with indeterminate data (where D*x* is **DB, DD, DF, DP, DQ, DT,** or **DW**).

[ ]
Ideal, MASM

expression1[*expression2*]

[*expression1*][*expression2*]

MASM mode: The [ ] operator can be used to specify addition or register indirect memory operands.

Ideal mode: The [ ] operator specifies a memory reference.

AND
Ideal, MASM

expression1 AND *expression2*

Performs a bit-by-bit logical AND of two expressions.

BYTE
Ideal

BYTE *expression*

Forces address expression to be byte size.

BYTE PTR
Ideal, MASM

BYTE PTR *expression*

Forces address expression to be byte size.

CODEPTR
Ideal, MASM

CODEPTR *expression*

Returns the default procedure address size.

DATAPTR
Ideal

DATAPTR *expression*

Forces address expression to model-dependent size.

DUP
Ideal, MASM

count DUP (*expression* [,*expression*]...)

Repeats a data allocation operation *count* times.

DWORD — Ideal

DWORD *expression*

Forces address expression to be doubleword size.

DWORD PTR — Ideal, MASM

DWORD PTR *expression*

Forces address expression to be doubleword size.

EQ — Ideal, MASM

expression1 EQ *expression2*

Returns true if expressions are equal.

FAR — Ideal

FAR *expression*

Forces an address expression to be a far code pointer.

FAR PTR — Ideal, MASM

FAR PTR *expression*

Forces an address expression to be a far code pointer.

FWORD — Ideal

FWORD *expression*

Forces address expression to be 32-bit far pointer size.

FWORD PTR — Ideal, MASM

FWORD PTR *expression*

Forces address expression to be 32-bit far pointer size.

GE — Ideal, MASM

expression1 GE *expression2*

Returns true if one expression is greater than or equal to the other.

GT — Ideal, MASM

expression1 GT *expression2*

Returns true if one expression is greater than the other.

HIGH — Ideal, MASM

HIGH *expression*

Returns the high part (8 bits or *type* size) of *expression*.

HIGH **Ideal**

type HIGH *expression*

Returns the high part (8 bits or *type* size) of *expression*.

LARGE **Ideal, MASM**

LARGE *expression*

Sets *expression*'s offset size to 32 bits. In Ideal mode, this operation is legal only if 386 code generation is enabled.

LE **Ideal, MASM**

expression1 LE *expression2*

Returns true if one expression is less than or equal to the other.

LENGTH **Ideal, MASM**

LENGTH *name*

Returns number of data elements allocated as part of *name*.

LOW **Ideal, MASM**

LOW *expression*

Returns the low part (8 bits or *type* size) of *expression*.

LOW **Ideal**

type LOW *expression*

Returns the low part (8 bits or *type* size) of *expression*.

LT **Ideal, MASM**

expression1 LT *expression2*

Returns true if one expression is less than the other.

MASK **Ideal, MASM**

MASK *recordfieldname*
MASK record

Returns a bit mask for a record field or an entire record.

MOD **Ideal, MASM**

expression1 MOD *expression2*

Returns remainder (modulus) from dividing two expressions.

NE
Ideal, MASM

expression1 NE *expression2*

Returns true if expressions are not equal.

NEAR
Ideal

NEAR *expression*

Forces an address expression to be a near code pointer.

NEAR PTR
Ideal, MASM

NEAR PTR *expression*

Forces an address expression to be a near code pointer.

NOT
Ideal, MASM

NOT *expression*

Performs a bit-by-bit complement (invert) of *expression*.

OFFSET
Ideal, MASM

OFFSET *expression*

Returns the offset of *expression* within the current segment (or the group that the segment belongs to, if using simplified segmentation directives or Ideal mode).

OR
Ideal, MASM

expression1 OR *expression2*

Performs a bit-by-bit logical OR of two expressions.

PROC
Ideal

PROC *expression*

Forces an address expression to be a near or far code pointer.

PROC PTR
Ideal, MASM

PROC PTR *expression*

Forces an address expression to be a near or far code pointer.

PTR
Ideal, MASM

type PTR *expression*

Forces address expression to have *type* size.

PWORD — Ideal

PWORD *expression*

Forces address expression to be 32-bit far pointer size.

PWORD PTR — Ideal, MASM

PWORD PTR *expression*

Forces address expression to be 32-bit far pointer size.

QWORD — Ideal

QWORD *expression*

Forces address expression to be quadword size.

QWORD PTR — Ideal, MASM

QWORD PTR *expression*

Forces address expression to be quadword size.

SEG — Ideal, MASM

SEG *expression*

Returns the segment address of an expression that references memory.

SHL — Ideal, MASM

expression SHL *count*

Shifts the value of *expression* to the left *count* bits. A negative count causes the data to be shifted the opposite way.

SHORT — Ideal, MASM

SHORT *expression*

Forces *expression* to be a short code pointer (within -128 to +127 bytes of the current code location).

SHR — Ideal, MASM

expression SHR *count*

Shifts the value of *expression* to the right *count* bits. A negative count causes the data to be shifted the opposite way.

SIZE — Ideal, MASM

SIZE *name*

Returns size of data item allocated with *name*. In MASM mode, **SIZE** returns the value of **LENGTH** *name* multiplied by **TYPE** *name*. In Ideal mode, **SIZE** returns the byte count within *name*'s **DUP**.

SMALL
Ideal, MASM

SMALL *expression*

Sets *expression*'s offset size to 16 bits. In Ideal mode, this operation is legal only if 386 code generation is enabled.

SYMTYPE
Ideal

SYMTYPE

Returns a byte describing *expression*.

TBYTE
Ideal

TBYTE *expression*

Forces address expression to be 10-byte size.

TBYTE PTR
Ideal, MASM

TBYTE PTR *expression*

Forces address expression to be 10-byte size.

THIS
Ideal, MASM

THIS *type*

Creates an operand whose address is the current segment and location counter. *type* describes the size of the operand and whether it refers to code or data.

.TYPE
MASM

.TYPE *expression*

Returns a byte describing the mode and scope of *expression*.

TYPE
IDEAL

TYPE *name1 name2*

Applies the type of an existing variable or structure member to another variable or structure member.

TYPE
MASM

TYPE *expression*

Returns a number indicating the size or type of *expression*.

UNKNOWN
Ideal

UNKNOWN *expression*

Removes type information from address expression.

WIDTH **Ideal, MASM**

WIDTH *recordfieldname*
WIDTH *record*

Returns the width in bits of a field in a record, or of an entire record.

WORD **Ideal**

WORD *expression*

Forces address expression to be word size.

WORD PTR **Ideal, MASM**

WORD PTR *expression*

Forces address expression to be word size.

XOR **Ideal, MASM**

expression1 XOR *expression2*

Performs bit-by-bit logical exclusive OR of two expressions.
Unconditional page break inserted for print formatting

The special macro operators

& **Ideal, MASM**

&name

Substitutes actual value of macro parameter *name.*

< > **Ideal, MASM**

Treats *text* literally, regardless of any special characters it might contain.

! **Ideal, MASM**

!character

Treats *character* literally, regardless of any special meaning it might otherwise have.

% **Ideal, MASM**

%text

Treats *text* as an expression, computes its value and replaces *text* with the result. *text* may be either a numeric expression or a text equate.

;; **Ideal, MASM**

;;comment

Suppresses storage of a comment in a macro definition.

P A R T 3

Directives

.186 MASM

Enables assembly of 80186 processor instructions.

.286 MASM

Enables assembly of non-privileged (real mode) 80286 processor instructions and 80287 numeric coprocessor instructions.

.286C MASM

Enables assembly of non-privileged (real mode) 80286 processor instructions and 80287 numeric coprocessor instructions.

.286P MASM

Enables assembly of all 80286 (including protected mode) processor instructions and 80287 numeric coprocessor instructions.

.287 MASM

Enables assembly of 80287 numeric coprocessor instructions.

.386 MASM

Enables assembly of non-privileged (real mode) 386 processor instructions and 387 numeric coprocessor instructions.

.386C MASM

Enables assembly of non-privileged (real mode) 386 processor instructions and 387 numeric coprocessor instructions.

.386P MASM

Enables assembly of all 386 (including protected mode) processor instructions and 387 numeric coprocessor instructions.

.387 MASM

Enables assembly of 387 numeric coprocessor instructions.

.486 MASM

Enables assembly of non-privileged (real mode) instructions for the i486 processor.

.486C MASM

Enables assembly of non-privileged (real mode) instructions for the i486 processor.

.486P MASM

Enables assembly of protected mode instructions for the 80486 processor.

.487 MASM

Enables assembly of 487 numeric processor instructions.

.586 MASM

Enables assembly of non-privileged (real mode) instructions for the Pentium processor.

.586C MASM

Enables assembly of non-privileged (real mode) instructions for the Pentium processor.

.586P MASM

Enables assembly of protected mode instructions for the Pentium processor.

.587 MASM

Enables assembly of Pentium numeric processor instructions.

.8086 MASM

Enables assembly of 8086 processor instructions only. This is the default processor instruction mode used by Turbo Assembler.

.8087 MASM

Enables assembly of 8087 numeric coprocessor instructions only. This is the default coprocessor instruction mode used by Turbo Assembler.

: Ideal, MASM

name:

Defines a near code label called *name*.

= Ideal, MASM

name = *expression*

Defines or redefines a numeric equate.

ALIGN Ideal, MASM

ALIGN *boundary*

Rounds up the location counter to a power-of-two address boundary (2, 4, 8, ...).

.ALPHA MASM

Sets alphanumeric segment-ordering. The **/a** command-line option performs the same function.

ALIAS

Ideal, MASM

ALIAS <alias_name>=<target_name>

Allows the association of an alias name with a particular target name. When the linker encounters an alias name, it resolves the alias by referring to the target name.

NOTE: The syntax for ALIAS is identical in both Ideal and MASM modes.

ARG

Ideal, MASM

ARG *argument* [,*argument*] ... [=*symbol*]
  [RETURNS *argument* [,*argument*]]

Sets up arguments on the stack for procedures. Each argument is assigned a positive offset from the BP register, presuming that both the return address of the procedure call and the caller's BP have been pushed onto the stack already. Each *argument* has the following syntax (boldface items are literal):

argname [[count1]] [:[debug_size] [type] [:count2]]

The optional *debug_size* has this syntax:

[type] **PTR**

ASSUME

Ideal, MASM

ASSUME *segmentreg*:*name* [,*segmentreg*:*name*]...
ASSUME *segmentreg*:NOTHING
ASSUME NOTHING

Specifies the segment register (*segmentreg*) that will be used to calculate the effective addresses for all labels and variables defined under a given segment or group name (*name*). The **NOTHING** keyword cancels the association between the designated segment register and segment or group name. The **ASSUME NOTHING** statement removes all associations between segment registers and segment or group names.

%BIN

Ideal, MASM

%BIN *size*

Sets the width of the object code field in the listing file to *size* columns.

CALL

Ideal, MASM

CALL<*instance_ptr*>METHOD{*object_name*>:}
  <*method_name*>{USES{*segreg*:}*offsreg*}{<*extended_call_parameters*>}

Calls a method procedure.

CATSTR

Ideal, MASM51

name CATSTR *string* [,*string*]...

Concatenates several strings to form a single string *name*.

.CODE — MASM

Same as **CODESEG**. MASM mode only.

CODESEG — Ideal, MASM

CODESEG [*name*]

Defines the start of a code segment when used with the **.MODEL** directive. If you have specified the medium or large memory model, you can follow the **.CODE** (or **CODESEG**) directive with an optional name that indicates the name of the segment.

COMM — Ideal, MASM

COMM *definition* [,*definition*]...

Defines a communal variable. Each definition describes a symbol and has the following format (boldface items are literal):

[*distance*] [*language*] *symbolname*[[*count1*]]:*type* [:*count2*]

distance can be either **NEAR** or **FAR** and defaults to the size of the default data memory model if not specified. *language* is either **C, PASCAL, BASIC, FORTRAN, PROLOG,** or **NOLANGUAGE** and defines any language-specific conventions to be applied to *symbolname*. *symbolname* is the communal symbol (or symbols, separated by commas). If *distance* is **NEAR**, the linker uses *count1* to calculate the total size of the array. If *distance* is **FAR**, the linker uses *count2* to indicate how many elements there are of size *count1* times the basic element size (determined by *type*). *type* can be one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name. *count2* specifies how many items this communal symbol defines. Both *count1* and *count2* default to 1.

COMMENT — MASM

COMMENT *delimiter* [*text*]
[*text*]
delimiter [*text*]

Starts a multiline comment. *delimiter* is the first non-blank character following **COMMENT**.

%COND — Ideal, MASM

Shows all statements in conditional blocks in the listing. This is the default mode for Turbo Assembler.

.CONST — MASM

Same as **CONST**. MASM mode only.

CONST — Ideal, MASM

Defines the start of the constant data segment.

.CREF — MASM

Same as **%CREF**. MASM mode only.

%CREF — Ideal, MASM

Allows cross-reference information to be accumulated for all symbols encountered from this point forward in the source file. **.CREF** reverses the effect of any **%XCREF** or **.XCREF** directives that inhibited the information collection.

%CREFALL — Ideal, MASM

Causes all subsequent symbols in the source file to appear in the cross-reference listing. This is the default mode for Turbo Assembler. **%CREFALL** reverses the effect of any previous **%CREFREF** or **%CREFUREF** directives that disabled the listing of unreferenced or referenced symbols.

%CREFREF — Ideal, MASM

Disables listing of unreferenced symbols in cross-reference.

%CREFUREF — Ideal, MASM

Lists only the unreferenced symbols in cross-reference.

%CTLS — Ideal, MASM

Causes listing control directives (such as **%LIST**, **%INCL**, and so on) to be placed in the listing file.

.DATA — MASM

Same as **DATASE6**. MASM mode only.

DATASEG — Ideal

Defines the start of the initialized data segment in your module. You must first have used the **.MODEL** directive to specify a memory model. The data segment is put in a group called DGROUP, which also contains the segments defined with the **.STACK**, **.CONST**, and **.DATA?** directives.

.DATA? — MASM

Defines the start of the uninitialized data segment in your module. You must first have used the **.MODEL** directive to specify a memory model. The data segment is put in a group called DGROUP, which also contains the segments defined with the **.STACK**, **.CONST**, and **.DATA** directives.

DB
Ideal, MASM

[*name*] DB *expression* [,*expression*]...

Allocates and initializes a byte of storage. *name* is the symbol you'll subsequently use to refer to the data. *expression* can be a constant expression, a question mark, a character string, or a **DUP**licated expression.

DD
Ideal, MASM

[*name*] DD [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 4 bytes (a doubleword) of storage. *name* is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. *expression* can be a constant expression, a 32-bit floating-point number, a question mark, an address expression, or a **DUP**licated expression.

%DEPTH
Ideal, MASM

%DEPTH *width*

Sets size of depth field in listing file to *width* columns. The default is 1 column.

DF
Ideal, MASM

[*name*] DF [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 6 bytes (a far 48-bit pointer) of storage. name is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. *expression* can be a constant expression, a question mark, an address expression, or a **DUP**licated expression.

DISPLAY
Ideal, MASM

DISPLAY "*text*"

Outputs a quoted string (*text*) to the screen.

DOSSEG
Ideal, MASM

Enables DOS segment-ordering at link time. **DOSSEG** is included for backward compatibility only.

DP

Ideal, MASM

[*name*] **DP** [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 6 bytes (a far 48-bit pointer) of storage. name is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. *expression* can be a constant expression, a question mark, an address expression, or a **DUP**licated expression.

DQ

Ideal, MASM

[*name*] DQ *expression* [,*expression*]...

Allocates and initializes 8 bytes (a quadword) of storage. *name* is the symbol you'll subsequently use to refer to the data. expression can be a constant expression, a 64-bit floating-point number, a question mark, or a **DUP**licated expression.

DT

Ideal, MASM

[*name*] DT *expression* [,*expression*]...

Allocates and initializes 10 bytes of storage. name is the symbol you'll subsequently use to refer to the data. *expression* can be a constant expression, a packed decimal constant expression, a question mark, an 80-bit floating-point number, or a **DUP**licated expression.

DW

Ideal, MASM

[*name*] DW [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 2 bytes (a word) of storage. *name* is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. expression can be a constant expression, a question mark, an address expression, or a **DUP**licated expression.

ELSE

Ideal, MASM

ELSE

```
IF condition
  statements1
[ELSE
  statements2]
ENDIF
```

Starts alternative conditional assembly block. The statements introduced by ELSE (*statements2*) are assembled if *condition* evaluates to false.

ELSEIF
Ideal, MASM

ELSEIF

```
IF condition1
 statements1
[ELSEIF condition2
 statements2]
ENDIF
```

Starts nested conditional assembly block if *condition2* is true. Several other forms of **ELSEIF** are supported: **ELSEIF1, ELSEIF2, ELSEIFB, ELSEIFDEF, ELSEIFDIF, ELSEIFDIFI, ELSEIFE, ELSEIFIDN, ELSEIFIDNI, ELSEIFNB,** and **ELSEIFNDEF.**

EMUL
Ideal, MASM

Causes all subsequent numeric coprocessor instructions to be generated as emulated instructions, instead of real instructions. When your program is executed, you must have a software floating-point emulation package installed or these instructions will not work properly.

END
Ideal, MASM

END [*startaddress*]

Marks the end of a source file. *startaddress* is a symbol or expression that specifies the address in your program where you want execution to begin. Turbo Assembler ignores any text that appears after the **END** directive.

ENDIF
Ideal, MASM

ENDIF

```
IF condition
 statements
ENDIF
```

Marks the end of a conditional assembly block started with one if the **IF***xxxx* directives.

ENDM
Ideal, MASM

Marks the end of a repeat block or a macro definition.

ENDP
Ideal, MASM

ENDP [*procname*]
[*procname*] ENDP

Marks the end of a procedure. If *procname* is supplied, it must match the procedure name specified with the **PROC** directive that started the procedure definition.

ENDS
Ideal, MASM

ENDS [*segmentname* | *strucname*]
[*segmentname* | *strucname*]ENDS

Marks end of current segment, structure or union. If you supply the optional name, it must match the name specified with the corresponding **SEGMENT**, **STRUC**, or **UNION** directive.

ENUM
Ideal, MASM

ENUM *name*[*enum_var*[,*enum_var*...]]
name ENUM [*enum_var*[,*enum_var*...]]

Declares an enumberated data type.

EQU
Ideal, MASM

name EQU *expression*

Defines *name* to be a string, alias, or numeric equate containing the result of evaluating *expression*.

.ERR
MASM

.ERR <*string*>

ERR
Ideal, MASM

ERR <*string*>

Forces an error to occur at the line that this directive is encountered on in the source file. The optional string will display as part of the error message.

.ERR1
MASM

.ERR1 <*string*>

Forces an error to occur on pass 1 of assembly. The optional string will display as part of the error message.

.ERR2
MASM

.ERR2 <*string*>

Forces an error to occur on pass 2 of assembly if multiple-pass mode (controlled by **/m** command-line option) is enabled. The optional string will display as part of the error message.

.ERRB
MASM

.ERRB *argument* <*string*>

Forces an error to occur if *argument* is blank (empty). The optional string will appear as part of the error message.

.ERRDEF MASM

.ERRDEF *symbol <string>*

Forces an error to occur if *symbol* is defined. The optional string will appear as part of the error message.

.ERRDIF MASM

.ERRDIF *argument1,argument2 <string>*

Forces an error to occur if arguments are different. The comparison is case sensitive. The optional string will appear as part of the error message.

.ERRDIFI MASM

.ERRDIFI *argument1,argument2 <string>*

Forces an error to occur if arguments are different. The comparison is not case sensitive. The optional string will appear as part of the error message.

.ERRE MASM

.ERRE *expression <string>*

Forces an error to occur if *expression* is false (0). The optional string will appear as part of the error message.

.ERRIDN MASM

.ERRIDN *argument1,argument2 <string>*

Forces an error to occur if arguments are identical. The comparison is case sensitive. The optional string will appear as part of the error message.

.ERRIDNI MASM

.ERRIDNI *argument1,argument2 <string>*

Forces an error to occur if arguments are identical. The comparison is not case sensitive. The optional string will appear as part of the error message.

ERRIF Ideal, MASM

ERRIF *expression <string>*

Forces an error to occur if *expression* is true (nonzero). The optional string will appear as part of the error message.

ERRIF1 **Ideal, MASM**

ERRIF1 <*string*>

Forces an error to occur on pass 1 of assembly. The optional string will appear as part of the error message.

ERRIF2 **Ideal, MASM**

ERRIF2 <*string*>

Forces an error to occur on pass 2 of assembly if multiple-pass mode (controlled by **/m** command-line option) is enabled. The optional string will appear as part of the error message.

ERRIFB **Ideal, MASM**

ERRIFB *argument* <*string*>

Forces an error to occur if *argument* is blank (empty). The optional string will appear as part of the error message.

ERRIFDEF **Ideal, MASM**

ERRIFDEF *symbol* <*string*>

Forces an error if *symbol* is defined. The optional string will appear as part of the error message.

ERRIFDIF **Ideal, MASM**

ERRIFDIF *argument1,argument2* <*string*>

Forces an error to occur if arguments are different. The comparison is case sensitive. The optional string will appear as part of the error message.

ERRIFDIFI **Ideal, MASM**

ERRIFDIFI *argument1,argument2* <*string*>

Forces an error to occur if arguments are different. The comparison is not case sensitive. The optional string will appear as part of the error message.

ERRIFE **Ideal, MASM**

ERRIFE *expression* <*string*>

Forces an error if *expression* is false (0). The optional string will appear as part of the error message.

ERRIFIDN
Ideal, MASM

ERRIFIDN *argument1,argument2 <string>*

Forces an error to occur if arguments are identical. The comparison is case sensitive. The optional string will appear as part of the error message.

ERRIFIDNI
Ideal, MASM

ERRIFIDNI *argument1,argument2 <string>*

Forces an error to occur if arguments are identical. The comparison is not case sensitive. The optional string will appear as part of the error message.

ERRIFNB
Ideal, MASM

ERRIFNB *argument <string>*

Forces an error to occur if *argument* is not blank. The optional string will appear as part of the error message.

ERRIFNDEF
Ideal, MASM

ERRIFNDEF *symbol <string>*

Forces an error to occur if *symbol* is not defined. The optional string will appear as part of the error message.

.ERRNB
MASM

.ERRNB *argument <string>*

Forces an error to occur if *argument* is not blank. The optional string will appear as part of the error message.

.ERRNDEF
MASM

.ERRNDEF *symbol <string>*

Forces an error to occur if *symbol* is not defined. The optional string will appear as part of the error message.

.ERRNZ
MASM

.ERRNZ *expression <string>*

Forces an error to occur if *expression* is true (nonzero). The optional string will appear as part of the error message.

EVEN
Ideal, MASM

Rounds up the location counter to the next even address.

EVENDATA

Ideal, MASM

Rounds up the location counter to the next even address in a data segment.

.EXIT

MASM

.EXIT [*return_value_expr*]

Produces termination code. MASM mode only. Equivalent to EXITCODE.

EXITCODE

Ideal, MASM

EXITCODE [*return_value_expr*]

Produces termination code. You can use it for each desired exit point. *return_value_expr* is a number to be returned to the operating system. If you don't specify *return_value_expr*, the value in AX is returned.

EXITM

Ideal, MASM

Terminates macro- or block-repeat expansion and returns control to the next statement following the macro or repeat-block call.

EXTRN

Ideal, MASM

EXTRN *definition* [*,definition*]...

Indicates that a symbol is defined in another module. *definition* describes a symbol and has the following format:

[*language*] *name*[*count1*]:*type* [:*count2*]

language specifies that the naming conventions of **C**, **PASCAL**, **BASIC**, **FORTRAN**, **ASSEMBLER**, or **PROLOG** are to be applied to symbol *name*. *name* is the symbol that is defined in another module and can optionally be followed by *count1*, an array element multiplier that defaults to 1. *type* must match the type of the symbol where it's defined and must be one of the following: **NEAR**, **FAR**, **PROC**, **BYTE**, **WORD**, **DWORD**, **DATAPTR**, **CODEPTR**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, **ABS**, or a structure name. *count2* specifies how many items this external symbol defines and defaults to 1 if not specified.

.FARDATA

MASM

Same as **FARDATA**. MASM mode only.

FARDATA

Ideal

FARDATA [*segmentname*]

Defines the start of a far initialized data segment. *segmentname*, if present, overrides the default segment name.

.FARDATA? MASM

.FARDATA? [*segmentname*]

Defines the start of a far uninitialized data segment. *segmentname,* if present, overrides the default segment name.

FASTIMUL Ideal, MASM

FASTIMUL*<dest_reg>,<source_r/m>,<value>*

Generates code that multiplies source register or memory address by value, and puts it into destination register.

FLIPFLAG Ideal, MASM

See syntax for the XOR processor instruction

Optimized form of XOR that complements bits with shortest possible instruction. Use only if the resulting contents of the flags registers are unimportant.

GETFIELD Ideal, MASM

GETFIELD*<field_name><destination_reg>,<source_r/m>*

Generates code that retrieves the value of a field found in the same source register or memory address, and sets the destination to that value.

GLOBAL Ideal, MASM

GLOBAL *definition* [,*definition*]...

Acts as a combination of the **EXTRN** and **PUBLIC** directives to define a global symbol. *definition* describes the symbol and has the following format (boldface items are literal):

[*language*] *name* [[*count1*]] :*type* [:*count2*]

language specifies that the naming conventions of **C, PASCAL, BASIC, FORTRAN, NOLANGUAGE,** or **PROLOG** are to be applied to symbol *name*. If *name* is defined in the current source file, it is made public exactly as if used in a **PUBLIC** directive. If not, it is declared as an external symbol of type *type,* as if the **EXTRN** directive had been used. *name* can be followed by an optional array count multiplier, *count1,* which defaults to 1. *type* must match the type of the symbol in the module where it is defined and must be one of the following: **NEAR, FAR, PROC, BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, ABS,** or a structure name. *count2* specifies how many items this symbol defines (1 is the default).

GOTO Ideal, MASM

GOTO *tag_symbol*

Tells Turbo Assembler to resume execution at the specified macro tag (*tag_symbol*). GOTO terminates any conditional block that it is found in.

GROUP

Ideal, MASM

GROUP *groupname segmentname* [,*segmentname*]...
groupname GROUP *segmentname* [,*segmentname*]...

Associates *groupname* with one or more segments, so that all labels and variables defined in those segments have their offsets computed relative to the beginning of group *groupname*. *segmentname* can be either a segment name defined previously with **SEGMENT** or an expression starting with **SEG**. In MASM mode, you must use a group override whenever you access a symbol in a segment that is part of a group. In Ideal mode, Turbo Assembler automatically generates group overrides for such symbols.

IDEAL

Ideal, MASM

Enters Ideal assembly mode. Ideal mode will stay in effect until it is overridden by a **MASM** or **QUIRKS** directive.

IF

Ideal, MASM

IF *expression*
 truestatements
[ELSE
 falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *expression* is true (nonzero).

IF1

Ideal, MASM

IF1
 truestatements
[ELSE
 falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the current assembly pass is pass one.

IF2

Ideal, MASM

IF2
 truestatements
[ELSE
 falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that multiple-pass mode (controlled by the **/m** command-line option) is enabled and the current assembly pass is pass two.

IFB

Ideal, MASM

```
IFB argument
 truestatements
[ELSE
 falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *argument* is blank (empty).

IFDEF

Ideal, MASM

```
IFDEF symbol
 truestatements
[ELSE
 falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *symbol* is defined.

IFDIF

Ideal, MASM

```
IFDIF argument1,argument2
 truestatements
[ELSE
 falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are different. The comparison is case sensitive.

IFDIFI

Ideal, MASM

```
IFDIFI argument1,argument2
 truestatements
[ELSE
 falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are different. The comparison is not case sensitive.

IFE
Ideal, MASM

```
IFE expression
  truestatements
[ELSE
  falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *expression* is false.

IFIDN
Ideal, MASM

```
IFIDN argument1,argument2
  truestatements
[ELSE
  falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are identical. The comparison is case sensitive.

IFIDNI
Ideal, MASM

```
IFIDNI argument1,argument2
  truestatements
[ELSE
  falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are identical. The comparison is not case sensitive.

IFNB
Ideal, MASM

```
IFNB argument
  truestatements
ELSE
  falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *argument* is nonblank.

IFNDEF

Ideal, MASM

```
IFNDEF symbol
 truestatements
[ELSE
 falsestatements]
ENDIF
```

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *symbol* is not defined.

%INCL

Ideal, MASM

Enables listing of include files. This is the default **INCLUDE** file listing mode.

INCLUDE

MASM, Ideal

INCLUDE *filename* or INCLUDE "*filename*"
Includes source code from file *filename* at the current position in the module being assembled. If no extension is specified, .ASM is assumed.

INCLUDELIB

MASM, Ideal

INCLUDELIB *filename* or INCLUDELIB "*filename*"
Causes the linker to include library *filename* at link time. If no extension is specified, .LIB is assumed.

INSTR

Ideal, MASM51

```
name INSTR [start,]string1,string2
```

name is assigned the position of the first instance of *string2* in *string1*. Searching begins at position *start* (position one if *start* not specified). If *string2* does not appear anywhere within *string1*, *name* is set to zero.

IRP

Ideal, MASM

```
IRP parameter,arg1[,arg2]...
 statements
ENDM
```

Repeats a block of statements with string substitution. *statements* are assembled once for each argument present. The arguments may be any text, such as symbols, strings, numbers, and so on. Each time the block is assembled, the next argument in the list is substituted for any instance of *parameter* in the *statements*.

IRPC

Ideal, MASM

IRPC *parameter,string*
  statements
ENDM

Repeats a block of statements with character substitution. *statements* are assembled once for each character in *string*. Each time the block is assembled, the next character in the string is substituted for any instances of *parameter* in *statements*.

JMP

Ideal, MASM

JMP<*instance_ptr*>METHOD{<*object_name*>:}
  <*method_name*>{USES{*segreg*:}*offsreg*}

Functions exactly like CALL..METHOD except that it generates a JMP instead of a CALL and it cleans up the stack if there are LOCAL or USES variables on the stack. Use primarily for tail recursion.

JUMPS

Ideal, MASM

Causes Turbo Assembler to look at the destination address of a conditional jump instruction, and if it is too far away to reach with the short displacement that these instructions use, it generates a conditional jump of the opposite sense around an ordinary jump instruction to the desired target address. This directive has the same effect as using the **/JJUMPS** command-line option.

LABEL

MASM, Ideal

name LABEL *type*
LABEL *name type*

Defines a symbol *name* to be of type *type*. *name* must not have been defined previously in the source file. *type* must be one of the following: **NEAR, FAR, PROC, BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE**, or a structure name.

.LALL

MASM

Enables listing of macro expansions.

LARGESTACK

Ideal, MASM

Indicates that the stack is 32 bit.

.LFCOND

MASM

Shows all statements in conditional blocks in the listing.

%LINUM
Ideal, MASM

%LINUM *size*

Sets the width of the line-number field in listing file to *size* columns. The default is four columns.

%LIST
Ideal, MASM

Shows source lines in the listing. This is the default listing mode.

.LIST
MASM

Same as **%LIST**. MASM mode only.

LOCAL
Ideal, MASM

In macros:
LOCAL *symbol* [,*symbol*]...

In procedures:
LOCAL *element* [,*element*]... [=*symbol*]

Defines local variables for macros and procedures. Within a macro definition, **LOCAL** defines temporary symbol names that are replaced by new unique symbol names each time the macro is expanded. **LOCAL** must appear before any other statements in the macro definition.

Within a procedure, **LOCAL** defines names that access stack locations as negative offsets relative to the BP register. If you end the argument list with an equal sign (=) and a symbol, that symbol will be equated to the total size of the local symbol block in bytes. Each *element* has the following syntax (boldface brackets are literal):

symname [[*count1*]] [:[*debug_size*] [:*type*] [:*count2*]]

type is the data type of the argument. It can be one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, NEAR, FAR, PROC**, or a structure name. If you don't specify a type, **WORD** size is assumed.

count2 specifies how many items of type the symbol defines. The default for *count2* is 1 if it is not specified.

count1 is an array element size multiplier. The total space allocated for the symbol is *count2* times the length specified by the *type* field times *count1*. The default for *count1* is 1 if it is not specified.

The optional *debug_size* has this syntax:

[*type*] **PTR**

LOCALS — Ideal, MASM

LOCALS [*prefix*]

Enables local symbols, whose names will begin with two at-signs (@@) or the two-character *prefix* if it is specified. Local symbols are automatically enabled in Ideal mode.

MACRO — Ideal, MASM

MACRO *name* [*parameter* [,*parameter*]...]
name MACRO [*parameter* [,*parameter*]...]

Defines a macro to be expanded later when *name* is encountered. *parameter* is a placeholder that you use in the the body of the macro definition wherever you want to substitute one of the actual arguments the macro is called with.

%MACS — Ideal, MASM

Enables listing of macro expansions.

MASKFLAG — Ideal, MASM

See the syntax for the AND processor instruction

Optimized form of AND that clears bits with the shortest possible instruction. Use only if the resulting contents of the flags registers are unimportant.

MASM — Ideal, MASM

Enters MASM assembly mode. This is the default assembly mode for Turbo Assembler.

MASM51 — Ideal, MASM

Enables assembly of some MASM 5.1 enhancements.

MODEL — Ideal, MASM

MODEL [*model modifier*] *memorymodel* [*module name*]
    [,[*language modifier*] *language*] [,*model modifier*]

Sets the memory model for simplified segmentation directives. *model modifier* can come before *memorymodel* or at the end of the statement and must be either **NEARSTACK** or **FARSTACK** if present. *memorymodel* is **TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE** or **TCHUGE.** *module name* is used in the large models to declare the name of the code segment. *language modifier* is **WINDOWS, ODDNEAR, ODDFAR,** or **NORMAL** and specifies generation of MSWindows procedure entry and exit code. *language* specifies which language you will be calling from to access the procedures in this module: **C, PASCAL, BASIC, FORTRAN,**

PROLOG, or **NOLANGUAGE**. Turbo Assembler automatically generates the appropriate procedure entry and exit code when you use the **PROC** and **ENDP** directives. *language* also tells Turbo Assembler which naming conventions to use for public and external symbols, and in what order procedure arguments were pushed onto the stack by the calling module. Also, the appropriate form of the **RET** instruction is generated to remove the arguments from the stack before returning if required.

.MODEL — MASM

Same as **MODEL**. MASM mode only.

MULTERRS — Ideal, MASM

Allows multiple errors to be reported on a single source line.

NAME — Ideal, MASM

NAME *modulename*

Sets the object file's module name. This directive has no effect in MASM mode; it only works in Ideal mode.

%NEWPAGE — Ideal, MASM

Starts a new page in the listing file.

%NOCONDS — Ideal, MASM

Disables the placement of statements in conditional blocks in the listing file.

%NOCREF — Ideal, MASM

%NOCREF [*symbol, ...*]

Disables cross-reference listing (CREF) information accumulation. If you supply one or more symbol names, cross-referencing is disabled only for those symbols.

%NOCTLS — Ideal, MASM

Disables placement of listing-control directives in the listing file. This is the default listing-control mode for Turbo Assembler.

NOEMUL — Ideal, MASM

Causes all subsequent numeric coprocessor instructions to be generated as real instructions, instead of emulated instructions. When your program is executed, you must have an 80x87 coprocessor installed or these instructions will not work properly. This is the default floating-point assembly mode for Turbo Assembler.

%NOINCL Ideal, MASM

Disables listing of source lines from **INCLUDE** files.

NOJUMPS Ideal, MASM

Disables stretching of conditional jumps enabled with **JUMPS**. This is the default mode for Turbo Assembler.

%NOLIST Ideal, MASM

Disables output to the listing file.

NOLOCALS Ideal, MASM

Disables local symbols enabled with **LOCALS**. This is the default for Turbo Assembler's MASM mode.

%NOMACS Ideal, MASM

Lists only macro expansions that generate code. This is the default macro listing mode for Turbo Assembler.

NOMASM51 Ideal, MASM

Disables assembly of certain MASM 5.1 enhancements enabled with **MASM51**. This is the default mode for Turbo Assembler.

NOMULTERRS Ideal, MASM

Allows only a single error to be reported on a source line. This is the default error-reporting mode for Turbo Assembler.

NOSMART Ideal, MASM

Disables code optimizations that generate different code than MASM.

%NOSYMS Ideal, MASM

Disables placement of the symbol table in the listing file.

%NOTRUNC Ideal, MASM

Prevents truncation of fields whose contents are longer than the corresponding field widths in the listing file.
60 points

NOWARN Ideal, MASM

NOWARN [*warnclass*]

Disables warning messages with warning identifier *warnclass*, or all warning messages if *warnclass* is not specified.

ORG
Ideal, MASM

ORG *expression*

Sets the location counter in the current segment to the address specified by *expression*.

%OUT
MASM

%OUT *text*

Displays *text* on screen.

P186
Ideal, MASM

Enables assembly of 80186 processor instructions.

P286
Ideal, MASM

Enables assembly of all 80286 (including protected mode) processor instructions and 80287 numeric coprocessor instructions.

P286N
Ideal, MASM

Enables assembly of non-privileged (real mode) 80286 processor instructions and 80287 numeric coprocessor instructions.

P286P
Ideal, MASM

Enables assembly of all 80286 (including protected mode) processor instructions and 80287 numeric coprocessor instructions.

P287
Ideal, MASM

Enables assembly of 80287 numeric coprocessor instructions.

P386
Ideal, MASM

Enables assembly of all 386 (including protected mode) processor instructions and 387 numeric coprocessor instructions.

P386N
Ideal, MASM

Enables assembly of non-privileged (real mode) 386 processor instructions and 387 numeric coprocessor instructions.

P386P
Ideal, MASM

Enables assembly of all 386 (including protected mode) processor instructions and 387 numeric coprocessor instructions.

P387
Ideal, MASM

Enables assembly of 387 numeric coprocessor instructions.

P486
Ideal, MASM

Enables assembly of all i486 (including protected mode) processor instructions.

P486N
Ideal, MASM

Enables assembly of non-privileged (real mode) i486 processor instructions.

P487
Ideal, MASM

Enables assembly of 487 numeric processor instructions.

P586
Ideal, MASM

Enables assembly of all Pentium (including protected mode) processor instructions.

P586N
Ideal, MASM

Enables assembly of non-privileged (real mode) Pentium processor instructions.

P587
Ideal, MASM

Enables assembly of Pentium numeric processor instructions.

P8086
Ideal, MASM

Enables assembly of 8086 processor instructions only. This is the default processor instruction mode for Turbo Assembler.

P8087
Ideal, MASM

Enables assembly of 8087 numeric coprocessor instructions only. This is the default coprocessor instruction mode for Turbo Assembler.
PAGE, MASM

%PAGESIZE
Ideal, MASM

PAGE [*rows*] [,*cols*]
%PAGESIZE [*rows*] [,*cols*]

Sets the listing page height and width, starts new pages. *rows* specifies the number of lines that will appear on each listing page (10..255). *cols* specifies the number of columns wide the page will be (59..255). Omitting *rows* or *cols* leaves the current setting unchanged. If you follow **PAGE** with a plus sign (+), a new page starts, the section number is incremented, and the page number restarts at 1. **PAGE** with no arguments forces the listing to resume on a new page, with no change in section number.

%PCNT
Ideal, MASM

%PCNT *width*

Sets segment:offset field width in listing file to *width* columns. The default is 4 for 16-bit segments and 8 for 32-bit segments.

PNO87
Ideal, MASM

Prevents the assembling of numeric coprocessor instructions (real or emulated).

%POPLCTL
Ideal, MASM

Resets the listing controls to the way they were when the last **%PUSHLCTL** directive was issued.

POPSTATE
Ideal, MASM

Returns to last saved state from Turbo Assembler's internal state stack.

PROC
Ideal, MASM

For VERSION T310 or earlier:
PROC [*language modifier*] [*language*] *name* [*distance*]
  [USES *items*,] [*argument* [,*argument*]...]
  [RETURNS *argument* [,*argument*]...]

For VERSION T320 or later:
PROC name [*language modifier*] [*language*] [*distance*]
  [USES *items*,] [*argument* [,*argument*]...]
  [RETURNS *argument* [,*argument*]...]
name PROC [*language modifier*] [*language*] [*distance*]
  [USES *items*,] [*argument* [,*argument*]...]
  [RETURNS *argument* [,*argument*]...]

Defines the start of procedure *name*. *language modifier* is either **WINDOWS** or **NOWINDOWS**, to specify generation of MSWindows entry/exit code. *language* specifies which language you will be calling from to access this procedure: **C, PASCAL, BASIC, FORTRAN, NOLANGUAGE**, or **PROLOG**. This determines symbol naming conventions, the order of any arguments on the stack, and whether the arguments will be left on the stack when the procedure returns. *distance* is **NEAR** or **FAR** and determines the type of **RET** instruction that will be assembled at the end of the procedure. *items* is a list of registers and/or single-token data items to be pushed on entry and popped on exit from the procedure. *argument* describes an argument the procedure is called with. Each *argument* has the following syntax:

argname[[*count1*]] [[:*distance*] **[PTR]** *type*] [:*count2*]

argname is the name you'll use to refer to this argument throughout the procedure. *distance* is **NEAR** or **FAR** to indicate that the argument is a pointer of the indicated size. *type* is the data type of the argument and can be **BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name. **WORD** is assumed if none is specified. *count1* and *count2* are the number of elements of type. **PTR** tells Turbo Assembler to emit debug information to let Turbo Debugger know that the argument is a pointer to a data item. Using **PTR** without *distance* causes the pointer size to be based on the current memory model and segment address size. **RETURNS** introduces one or more arguments that won't be popped from the stack when the procedure returns.

PROCDESC

Ideal, MASM

PROCDESC name [*language*] [*language modifier*] [*distance*]
[*arguments*]
name PRODESC [[*language_modifier*] *language*] [*distance*]
[*arguments*]

Declares a procedure prototype, which lets Turbo Assembler check the types and number of parameters to procedure calls and declarations, and specifies language and distance. Also serves to PUBLIC or EXTRN the procedure name.

PROCTYPE

Ideal, MASM

PROCTYPE name [*procedure_description*]
name PROCTYPE [*procedure_description*]

procedure_description has the following syntax:
[[*language_modifier*]*language*][*distance*][*argument_list*]
argument_list has the following syntax:
argument[,*argument*]...
where each argument has the following syntax:
[*argname*][[*count1_expressions*]]:*complex_type*[:*count2_expression*]

Declares a procedure type. Describes a procedure but does not create a prototype for it. Can be used in place of the language specifier in a call to allow argument type checking during compilation.

PUBLIC

Ideal, MASM

PUBLIC [*language*] *symbol* [,[*language*] *symbol*]...

Declares *symbol* to be accessible from other modules. If *language* is specified (**C**, **PASCAL**, **BASIC**, **FORTRAN**, **ASSEMBLER**, or **PROLOG**), *symbol* is made public after having the naming conventions of the specified language applied to it.

PUBLICDLL

Ideal, MASM

PUBLICDLL [*language*] *symbol* [,[*language*] *symbol*]...

Declares symbols to be accessible as dynamic link entry points from other modules. *symbol* (a **PROC** or program label, data variable name, or numeric constant defined with **EQU**) becomes accessible to other programs under Windows. If *language* is specified (**C**, **PASCAL**, **BASIC**, **FORTRAN**, **PROLOG**, or **NOLANGUAGE**), *symbol* is made public after having the naming conventions of the specified language applied to it.

PURGE — Ideal, MASM

PURGE *macroname* [,*macroname*]...

Removes macro definition *macroname*.

%PUSHLCTL — Ideal, MASM

Saves current listing controls on a 16-level stack.

PUSHSTATE — Ideal, MASM

Saves current operating state on an internal stack that is 16 levels deep.

QUIRKS — Ideal, MASM

Allows you to assemble a source file that makes use of one of the true MASM bugs.

.RADIX — MASM

Same as **RADIX**. MASM mode only.

RADIX — Ideal, MASM

RADIX *radix*

Sets the default radix for integer constants in expressions to 2, 8, 10, or 16.

RECORD — MASM, Ideal

name RECORD *field* [,*field*]...
RECORD *name field* [,*field*]...

Defines record *name* that contains bit fields. Each *field* describes a group of bits in the record and has the following format (boldface items are literal):

fieldname:*width*[=*expression*]

fieldname is the name of a field in the record. *width* (1..16) specifies the number of bits in the field. If the total number of bits in all fields is 8 or less, the record will occupy 1 byte; 9..16 bits will occupy 2 bytes; otherwise, it will occupy 4 bytes. *expression* provides a default value for the field.

REPT — Ideal, MASM

```
REPT expression
  statements
ENDM
```

Repeats a block of statements *expression* times.

RETCODE
Ideal, MASM

Generates either a near return (2-byte displacement) or a far return (4-byte displacement) depending on the size of the memory model declared in the .MODULE directive. A tiny, small, or compact memory model results in a near return, while a medium, large, or huge memory model results in a far return. See the RET processor instruction in Part 4 for more information.

RETF
Ideal, MASM

Generates a far return (4-byte displacement) from a procedure. See the RET processor instruction in Part 4 for more information.

RETN
Ideal, MASM

Generates a near return (2-byte displacement) from a procedure. See the RET processor instruction in Part 4 for more information.

.SALL
MASM

Suppresses the listing of all statements in macro expansions.

SEGMENT
MASM, Ideal

SEGMENT *name* [*align*] [*combine*] [*use*] [*'class'*]
name SEGMENT [*align*] [*combine*] [*use*] [*'class'*]

Defines segment *name* with full attribute control. If you have already defined a segment with the same name, this segment is treated as a continuation of the previous one. *align* specifies the type of memory boundary where the segment must start: **BYTE, WORD, DWORD, PARA** (default), or **PAGE**. *combine* specifies how segments from different modules but with the same name will be combined at link time: **AT** *expression* (locates segment at absolute paragraph address *expression*), **COMMON** (locates this segment and all other segments with the same name at the same address), **MEMORY** (concatenates all segments with the same name to form a single contiguous segment), **PRIVATE** (does not combine this segment with any other segments; this is the default used if none specified), **PUBLIC** (same as **MEMORY** above), **STACK** (concatenates all segments with the same name to form a single contiguous segment, then initializes SS to the beginning of the segment and SP to the length of the segment) or **VIRTUAL** (defines a special kind of segment that will be treated as a common area and attached to another segment at link time). *use* specifies the default word size for the segment if 386 code generation is enabled, and can be either **USE16** or **USE32**. *class* controls the ordering of segments at link time: segments with the same class name are loaded into memory together, regardless of the order in which they appear in the source file.

.SEQ
MASM

Sets sequential segment-ordering. This is the default ordering mode for Turbo Assembler. **.SEQ** has the same function as the **/s** command-line option.

SETFIELD
Ideal, MASM

SETFIELD<*field_name*><*destination_r/m*>,<*source_reg*>

Generates code that sets a value in a record field. Sets the field in the destination register or memory address with the contents of a source register.

SETFLAG
Ideal, MASM

see the syntax for the OR processor instruction

Optimized form of OR that sets bits with shortest possible instruction. Use only if the resulting contents of the flags register is unimportant.

.SFCOND
MASM

Prevents statements in false conditional blocks from appearing in the listing file.

SIZESTR
Ideal, MASM51

name SIZESTR *string*

Assigns the number of characters in *string* to *name*. A null string has a length of zero.

SMALLSTACK
Ideal, MASM

Indicates that the stack is 16 bit.

SMART
Ideal, MASM

Enables all code optimizations.

.STACK
MASM

Same as **STACK**. MASM mode only.

STACK
Ideal, MASM

STACK [*size*]

Defines the start of the stack segment, allocating *size* bytes. 1024 bytes are allocated if *size* is not specified.

.STARTUP
MASM

Provides initialization code. MASM mode only. Equivalent to **STARTUPCODE.**

STARTUPCODE
Ideal, MASM

Provides initialization code and marks the beginning of the program.

STRUC
Ideal, MASM

[*name*] STRUC{<*modifiers*>}{<*parent_name*>}{METHOD<*method_list*>}
<*structure_data*>
ENDS [*name*]

STRUC [*name*]{<*modifiers*>}{<*parent_name*>}{METHOD<*method_list*>}
<*structure_data*>
ENDS [*name*]

parent_name is the name of the parent object's data structure. *method_list* is like that of **TABLE**. *structure_data* is any (additional) data present in an instance of the object. *modifiers* can be GLOBAL, NEAR, or FAR.

SUBSTR
Ideal, MASM51

name SUBSTR *string,position*[*,size*]

Defines a new string *name* consisting of characters from *string* starting at *position,* with a length of *size*. All the remaining characters in *string,* starting from *position,* are assigned to *name* if *size* is not specified.

SUBTTL
MASM

Same as %**SUBTTL**. MASM mode only.

%SUBTTL
Ideal, MASM

%SUBTTL "*text*"

Sets subtitle in listing file to *text*.

%SYMS
Ideal, MASM

Enables symbol table placement in listing file. This is the default symbol listing mode for Turbo Assembler.

TABLE
Ideal, MASM

TABLE *name* [*table_member* [*,table_member...*]]

Constructs a table structure used to contain method pointers for objects.

%TABSIZE
Ideal, MASM

%TABSIZE *width*

Sets the number of columns between tabs in the listing file to *width*. The default is 8 columns.

TBLINIT
Ideal, MASM

Initializes pointer in an object to the virtual method table.

TBLINST

Ideal, MASM

TBLINST

Creates an instance of the virtual table for the current object and defines @TableAddr_<*object*>. Must be used after every object definition that includes virtual methods, so that the virtual table is allocated. You should use this directive in only one module of your program.

TBLPTR

Ideal, MASM

TBLPTR

Places a virtual table pointer within the object data. Defines a structure member of the name @Mptr_<*object*>. This can only be used inside an object definition.

TESTFLAG

Ideal, MASM

See the syntax for the TEST processor instruction

Optimized form of TEST that tests bits with the shortest possible instruction.

%TEXT

Ideal, MASM

%TEXT *width*

Sets width of source field in listing file to *width* columns.

.TFCOND

MASM

Toggles conditional block-listing mode.

TITLE

MASM

Same as **%TITLE**. MASM mode only.

%TITLE

Ideal, MASM

%TITLE "*text*"

Sets title in listing file to *text*.

%TRUNC

Ideal, MASM

Truncates listing fields that are too long.

TYPEDEF

Ideal, MASM

TYPEDEF *type_name complex_type*
type_name TYPEDEF *complex_type*

Defines named types.

UDATASEG

Ideal, MASM

Defines the start of an uninitialized data segment.

UFARDATA

Ideal, MASM

Defines the start of an uninitialized far data segment.

UNION

Ideal, MASM (disabled by QUIRKS)

UNION *name*
 fields
ENDS [*name*]
name UNION
 fields
[*name*] ENDS

Defines a union called *name*. A union is just like a **STRUC** except that all its members have an offset of zero from the start of the union. This results in a set of fields that are overlayed, allowing you to refer to the memory area defined by the union with different names and different data sizes. The length of a union is the length of its largest member, not the sum of the lengths of its members as in a **STRUC**. *fields* define the fields that comprise the union. Each field uses the normal data allocation directives (**DB**, **DW**, and so on) to define its size.

USES

Ideal, MASM

USES *item* [,*item*]...

Indicates which registers or single-token data items you want to have pushed at the beginning of the enclosing procedure and which ones you want popped just before the procedure returns. You must use this directive before the first instruction that actually generates code in your procedure.

VERSION

MASM, Ideal

VERSION <*version_ID*>

Places Turbo Assembler in the equivalent operating mode for the specified version.

WARN

Ideal, MASM

WARN [*warnclass*]

Enables the type of warning message specified with *warnclass*, or all warnings if *warnclass* is not specified. *warnclass* may be one of: **ALN**, **ASS**, **BRK**, **ICG**, **LCO**, **OPI**, **OPP**, **OPS**, **OVF**, **PDC**, **PRO**, **PQK**, **RES**, or **TPI**.

WHILE
Ideal, MASM

```
WHILE while_expression
  macro body
ENDM
```

Repeats a macro body until *while_expression* evaluates to 0 (false).

.XALL
MASM

Causes only macro expansions that generate code or data to be listed.

.XCREF
MASM

Disables cross-reference listing (CREF) information accumulation.

.XLIST
MASM

Disables subsequent output to listing file.

P A R T 4

Processor instructions

This part presents instructions for the x86 in alphabetical order. For each instruction, the forms are given for each operand combination, including object code produced, operands required, execution time, and a description. For each instruction, there is an operational description and a summary of exceptions generated.

Operand-size and address-size attributes

When executing an instruction, the x86 can address memory using either 16- or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. Sixteen-bit addresses imply both the use of a 16-bit displacement in the instruction and the generation of a 16-bit address offset (segment relative address) as the result of the effective address calculation. Thirty-two-bit addresses imply the use of a 32-bit displacement and the generation of a 32-bit address offset. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

Default segment attribute

For programs executed in protected mode, the D-bit in executable-segment descriptors determines the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A value of zero in the D-bit sets the default address size and operand size to 16 bits; a value of one, to 32 bits.

Programs that execute in real mode or virtual-8086 mode have 16-bit addresses and operands by default.

Operand-size and address-size instruction prefixes

The internal encoding of an instruction can include two byte-long prefixes: the address-size prefix, 67H, and the operand-size prefix, 66H. (A later section, "Instruction format," shows the position of the prefixes in an instruction's encoding.) These prefixes *override* the default segment attributes for the instruction that follows. Table 4.1 shows the effect of each possible combination of defaults and overrides.

Table 4.1 Effective size attributes

Segment default D= ...	0	0	0	0	1	1	1	1
Operand-size prefix 66h	N	N	Y	Y	N	N	Y	Y
Address-size prefix 67h	N	Y	N	Y	N	Y	N	Y
Effective operand size	16	16	32	32	32	32	16	16
Effective address size	16	32	16	32	32	16	32	16

Y = Yes, this instruction prefix is present.
N = No, this instruction prefix is *not* present.

Address-size attribute for stack

Instructions that use the stack implicitly (for example, POP EAX) also have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack.

The stack address-size attribute is controlled by the B-bit of the data-segment descriptor in the SS register. A value of zero in the B-bit selects a stack address-size attribute of 16; a value of one selects a stack address-size attribute of 32.

Instruction format

All instruction encodings are subsets of the general instruction format shown in Figure 4.1. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (scale index base) byte, a displacement, if required, and an immediate data field, if required.

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to a operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Figure 4.1
386 instruction format

Instruction prefix	Address-size prefix	Operand-size prefix	Segment override
0 or 1	0 or 1	0 or 1	0 or 1
Number of bytes			

Opcode	Modr/M	SIB	Displacement	Immediate
1 or 2	0 or 1	0 or 1	0, 1, 2, or 4	0, 1, 2, or 4
Number of bytes				

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8, 16, or 32 bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

- The following are the allowable instruction prefix codes:
- F3h: REP prefix (used only with string instructions)
- F3h: REPE/REPZ prefix (used only with string instructions)
- F2h: REPNE/REPNZ prefix (used only with string instructions)
- F0h: LOCK prefix

The following are the segment override prefixes:

- 2Eh: CS segment override prefix
- 36h: SS segment override prefix
- 3Eh: DS segment override prefix
- 26h: ES segment override prefix
- 64h: FS segment override prefix (386 processors and greater)
- 65h: GS segment override prefix (386 processors and greater)
- 66h: Operand-size override
- 67h: Address-size operand

ModR/M and SIB bytes

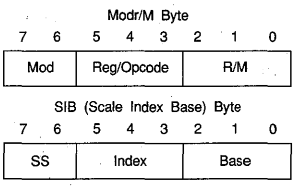
The ModR/M and SIB bytes follow the opcode byte(s) in many of the x86 instructions. They contain the following information: the indexing type or register number to be used in the instruction; the register to be used, or more information to select the instruction; and the base, index, and scale information.

The ModR/M byte contains three fields of information:

- The **mod** field, which occupies the two most significant bits of the byte, combines with the *r/m* field to form 32 possible values: 8 registers and 24 indexing modes.
- The **reg** field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The **r/m** field, which occupies the three least-significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the **mod** field as described earlier.
- The based indexed and scaled indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:
- The **ss** field, which occupies the 2 most-significant bits of the byte, specifies the scale factor.
- The **index** field, which occupies the next 3 bits following the **ss** field specifies the register number of the index register.
- The **base** field, which occupies the 3 least-significant bits of the byte, specifies the register number of the base register.

Figure 4.2 shows the format of the ModR/M and SIB bytes.

Figure 4.2
ModR/M and SIB byte formats



The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Tables 4.2, 4.3, and 4.4.

Table 4.2 16-bit addressing forms with ModR/M byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit (opcode)	0	1	2	3	4	5	6	7
REG =	000	001	010	011	100	101	110	111

Effective address	**ModR/M**		**ModR/M values in hexadecimal**							
[BX + SI]		000	00	08	10	18	20	28	30	38
[BX + DI]		001	01	09	11	19	21	29	31	39
[BP + SI]		010	02	0A	12	1A	22	2A	32	3A
[BP + DI]	00	011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX + SI] + disp8		000	40	48	50	58	60	68	70	78
[BX + DI] + disp8		001	41	49	51	59	61	69	71	79
[BP + SI] + disp8		010	42	4A	52	5A	62	6A	72	7A
[BP + DI] + disp8	01	011	43	4B	53	5B	63	6B	73	7B
[SI] + disp8		100	44	4C	54	5C	64	6C	74	7C
[DI] + disp8		101	45	4D	55	5D	65	6D	75	7D
[BP] + disp8		110	46	4E	56	5E	66	6E	76	7E
[BX] + disp8		111	47	4F	57	5F	67	6F	77	7F
[BX + SI] + disp16		000	80	88	90	98	A0	A8	B0	B8
[BX + DI] + disp16		001	81	89	91	99	A1	A9	B1	B9
[BP + SI] + disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP + DI] + disp16	10	011	83	8B	93	9B	A3	AB	B3	BB
[SI] + disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI] + disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP] + disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX] + disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL (386)		000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL (386)		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL (386)		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL (386)	11	011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH (386)		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH (386)		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH (386)		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH (386)		111	C7	CF	D7	DF	E7	EF	F7	FF

disp8 denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index. disp16 denotes a 16-bit displacement following the ModR/M byte, to be added to the index. Default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.

Table 4.3 32-bit addressing forms with ModR/M byte (386 only)

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit(opcode)	0	1	2	3	4	5	6	7
REG =	000	001	010	011	100	101	110	111

Effective address	**ModR/M**		**ModR/M values in hexadecimal**							
[EAX]		000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]	00	011	03	0B	13	1B	23	2B	33	3B
[- -] [- -]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX]		000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EPX];	01	011	43	4B	53	5B	63	6B	73	7B
disp8[- -] [- -]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]		000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]	10	011	83	8B	93	9B	A3	AB	B3	BB
disp32[- -] [- -]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL		000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL	11	011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

[- -] [- -] means a SIB follows the ModR/M byte. *disp8* denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index. *disp32* denotes a 32-bit displacement following the ModR/M byte, to be added to the index.

Table 4.4 32-bit addressing forms with SIB byte (386 only)

r32	EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
Base =	0	1	2	3	4	5	6	7
Base =	000	001	010	011	100	101	110	111

Scaled index	**SS index**		**ModR/M values in hexadecimal**							
[EAX]		000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]	00	011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]		000	40	41	42	44	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	55	54	55	56	57
[EBX*2]	01	011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]		000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]	10	011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]		000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]	11	011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

[*] means a disp32 with no base if MOD is 00; otherwise, [ESP].
This provides the following addressing modes:

disp32[index]	(MOD=00)
disp8[EBP][index]	(MOD=01)
disp32[EBP][index]	(MOD=10)

How to read the instruction set pages

Here's a sample of the format of this chapter:

Instruction name

What the instruction name means
What processor the instruction works on

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

Flag information goes here
```

Opcode Instruction	Clocks		
	386	286*	86
	This table contains clock information		

*Because the 186 processor is effectively a 286 without protected mode instructions, the 186 timings are identical to the timings listed for the 286.

Flags

Each entry in this section includes information on which flags in the x86's flag register are changed and how. Each flag has a one-letter tag for its name.

O = Overflow flag	**Z** = Zero flag
D = Direction flag	**A** = Auxiliary flag
I = Interrupt flag	**P** = Parity flag
T = Trap flag	**C** = Carry flag
S = Sign flag	

The following symbols indicate how the flag register has changed:

? = Undefined after the operation

***** = Changed to reflect the results of the instruction

0 = Always cleared

1 = Always set

Opcode

The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit
(digit is between 0 and 7.) Indicates that the ModR/M byte of the instruction uses only the *r/m* (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

/r
Indicates that the ModR/M byte of the instruction contains both a register operand and an *r/m* operand.

cb, cw, cd, cp
A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, iw, id
A 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes, or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, +rw, +rd
A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are

rb	**rw**	**rd** (386)
AL = 0	AX = 0	EAX = 0
CL = 1	CX = 1	ECX = 1
DL = 2	DX = 2	EDX = 2
BL = 3	BX = 3	EBX = 3
AH = 4	SP = 4	ESP = 4
AH = 4	SP = 4	ESP = 4
CH = 5	BP = 5	EBP = 5
DH = 6	SI = 6	ESI = 6
BH = 7	DI = 7	EDI = 7

Instruction

The "Instruction" column gives the syntax of the instruction statement as it would appear in a TASM 386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8
A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

rel16, rel32
A relative address within the same code segment as the instruction assembled. **rel16** applies to instructions with an operand-size attribute of 16 bits; **rel32** applies to instructions with an operand-size attribute of 32 bits (386 only).

ptr16:16, ptr16:32
A far pointer, typically in a code segment different from that of the instruction. The notation **16:16** indicates that the value of the pointer has two parts. The value to the right of the colon is a 16-bit selector or value destined for the code segment register. The value to the left corresponds to the offset within the destination segment. **ptr16:16** is used when the instruction's operand-size attribute is 16 bits; **ptr16:32** is used with the 32-bit attribute (386 only).

r8
One of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16
One of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32 (386)
One of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8
An immediate byte value. **imm8** is a signed number between -128 and +127 inclusive. For instructions in which **imm8** is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16
An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32,768 and +32,767 inclusive.

imm32 (386)
An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and -2,147,483,648.

r/m8
A 1-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16
A word register or memory operand used for instructions whose oper-

and-size attribute is 16 bits. The word registers are AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32
A doubleword register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword registers are EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m8
A memory byte addressed by DS:SI or ES:DI (used only by string instructions on the 386).

m16
A memory word addressed by DS:SI or ES:DI (used only by string instructions).

m32
A memory doubleword addressed by DS:SI or ES:DI (used only by string instructions).

m16:16, m16:32 (386)
A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

m16 & 32, m16 & 16 (186/286/386), m32 & 32 (386)
A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. **m16 & 16** and **m32 & 32** operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. **m16 & 32** is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

moffs8, moffs16, moffs32 (memory offset; 386 only)
A simple memory variable of type **BYTE, WORD,** or **DWORD** (386) used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with **moffs** indicates its size, which is determined by the address-size attribute of the instruction.

Sreg
A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4 (386), and GS = 5 (386).

Clocks

The "Clocks" column gives the number of clock cycles the instruction takes to execute. The clock count calculations make the following assumptions:

- The instruction has been prefetched and decoded and is ready for execution.
- Bus cycles do not require wait states.
- There are no local bus HOLD requests delaying processor access to the bus.
- No exceptions are detected during instruction execution.
- Memory operands are aligned.

Clock counts for instructions that have an *r/m* (register or memory) operand are separated by a slash. The count to the left is used for a register operand; the count to the right is used for a memory operand.

The following symbols are used in the clock count specifications:

- **n**, which represents a number of repetitions.
- **m**, which represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and every other byte of the instruction and prefix(es) each counts as one component.
- **pm=**, a clock count that applies when the instruction executes in protected mode. **pm=** is not given when the clock counts are the same for protected and real address modes.

When an exception occurs during the execution of an instruction and the exception handler is in another task, the instruction exception time is increased by the number of clocks to effect a task switch. This parameter depends on several factors:

- The type of TSS used to represent the current task (386 TSS or 286 TSS).
- The type of TSS used to represent the new task.
- Whether the current task is in V86 mode.
- Whether the new task is in V86 mode.

Note: Users should read Intel's documentation for more information about protected mode and task switching.

AAA ASCII adjust after addition

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
37	AAA	3	4	3	8	ASCII adjust after addition

Execute AAA only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the operands of the ADD instruction should be in the range 0 through 9 (BCD digits). In this case, AAA adjusts AL to contain the correct decimal digit result. If the addition produced a decimal carry, the AH register is incremented, and the carry and auxiliary carry flags are set to 1. If there was no decimal carry, the carry and auxiliary flags are set to 0 and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAA instruction with OR AL, 30H.

AAD ASCII adjust before division

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D5 0A	AAD	14	19	14	60	ASCII adjust before division

AAD is used to prepare two unpacked BCD digits (the least-significant digit in AL, the most-significant digit in AH) for a division operation that will yield an unpacked result. This is accomplished by setting AL to AL + (10 * AH), and then setting AH to 0. AX is then equal to the binary equivalent of the original unpacked two-digit number.

AAM ASCII adjust AX after multiply

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D4 0A	AAM	15	17	16	83	ASCII adjust AX after multiply

Execute AAM only after executing a MUL instruction between two unpacked BCD digits that leaves the result in the AX register. Because the result is less than 100, it is contained entirely in the AL register. AAM unpacks the AL result by dividing AL by 10, leaving the quotient (most-significant digit) in AH and the remainder (least-significant digit) in AL.

AAS ASCII adjust AL after subtraction

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
3F	AAS	3	4	3	8	ASCII adjust AL after subtraction

Execute AAS only after a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the operands of the SUB instruction must have been in the range 0 through 9 (BCD digits). In this case, AAS adjusts AL so it contains the correct decimal digit result. If the subtraction produced a decimal carry, the AH register is decremented, and the carry and auxiliary carry flags are set to 1. If no decimal carry occurred, the carry and auxiliary carry flags are set to 0, and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAS with OR AL, 30H.

ADC Add with carry

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
10 /r	ADC r/m8,r8	1/3	2/7	2/7	3/16+EA	Add with carry byte register to r/m byte
11 /r	ADC r/m16,r16	1/3	2/7	2/7	3/16+EA	Add with carry word register to r/m word
11 /r	ADC r/m32,r32	1/3	2/7			Add with CF dword register to r/m word
12 /r	ADC r8,r/m8	1/2	2/6	2/7	3/9+EA	Add with carry r/m byte to byte register
13 /r	ADC r16,r/m16	1/2	2/6	2/7	3/9+EA	Add with carry r/m word to word register
13 /r	ADC r32,r/m32	1/2	2/6			Add with CF r/m dword to dword register
14 ib	ADC AL,imm8	1	2	3	4	Add with carry immediate byte to AL
15 iw	ADC AX,imm16	1	2	3	4	Add with carry immediate word to AX
15 id	ADC EAX,imm32	1	2			Add with carry immediate dword to EAX
80 /2 ib	ADC r/m8,imm8	1/3	2/7	3/7	4/17+EA	Add with carry immediate byte to r/m byte
81 /2 iw	ADC r/m16,imm16	1/3	2/7	3/7	4/17+EA	Add with carry immediate word to r/m word
81 /2 id	ADC r/m32,imm32	1/3	2/7			Add with CF immediate dword to r/m dword
83 /2 ib	ADC r/m16,imm8	1/3	2/7	3/7	4/17+EA	Add with CF sign-extended immediate byte to r/m word
83 /2 ib	ADC r/m32,imm8	1/3	2/7			Add with CF sign-extended immediate byte into r/m dword

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

ADD Add

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
04 ib	ADD AL,imm8	1	2	3	4	Add immediate byte to AL
05 iw	ADD AX,imm16	1	2	3	4	Add immediate word to AX
05 id	ADD EAX,imm32	1	2			Add immediate dword to EAX
80 /0 ib	ADD r/m8,imm8	1/3	2/7	3/7	4/17+EA	Add immediate byte to r/m byte
81 /0 iw	ADD r/m16,imm16	1/3	2/7	3/7	4/17+EA	Add immediate word to r/m word
81 /0 id	ADD r/m32,imm32	1/3	2/7			Add immediate dword to r/m dword
83 /0 ib	ADD r/m16,imm8	1/3	2/7	3/7	4/17+EA	Add sign-extended immediate byte to r/m word
83 /0 ib	ADD r/m32,imm8	1/3	2/7			Add sign-extended immediate byte to r/m dword
00 /r	ADD r/m8,r8	1/3	2/7	2/7	3/16+EA	Add byte register to r/m byte
01 /r	ADD r/m16,r16	1/3	2/7	2/7	3/16+EA	Add word register to r/m word
01 /r	ADD r/m32,r32	1/3	2/7			Add dword register to r/m dword
02 /r	ADD r8,r/m8	1/2	2/6	2/7	3/9+EA	Add r/m byte to byte register
03 /r	ADD r16,r/m16	1/2	2/6	2/7	3/9+EA	Add r/m word to word register
03 /r	ADD r32,r/m32	1/2	2/6			Add r/m dword to dword register

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

AND Logical AND

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
20 /r	AND r/m8,r8	1/3	2/7	2/7	3/16+EA	AND byte register into r/m byte
21 /r	AND r/m16,r16	1/3	2/7	2/7	3/16+EA	AND word register into r/m word
21 /r	AND r/m32,r32	1/3	2/7			AND dword register to r/m dword
22 /r	AND r8,r/m8	1/2	2/6	2/7	3/9+EA	AND r/m byte to byte register
23 /r	AND r16,r/m16	1/2	2/6	2/7	3/9+EA	AND r/m word to word register
23 /r	AND r32,r/m32	1/2	2/6			AND r/m dword to dword register
24 ib	AND AL,imm8	1	2	3	4	AND immediate byte to AL
25 iw	AND AX,imm16	1	2	3	4	AND immediate word to AX
25 id	AND EAX,imm32	1	2			AND immediate dword to EAX

Opcode	Instruction	Clocks				Description
		486	386	286	86	
80 /4 ib	AND r/m8,imm8	1/3	2/7	3/7	4/17+EA	AND immediate byte to r/m byte
81 /4 iw	AND r/m16,imm16	1/3	2/7	3/7	4/17+EA	AND immediate word to r/m word
81 /4 id	AND r/m32,imm32	1/3	2/7			AND immediate dword to r/m word
83 /4 ib	AND r/m16,imm8	1/3	2/7	3/7	4/17+EA	AND sign-extended immediate byte with r/m word
83 /4 ib	AND r/m32,imm8	1/3	2/7			AND sign-extended immediate byte with r/m dword

Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

The optimized form of AND is MASKFLAG (see Part 3).

ARPL

Adjust RPL field of selector
80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks			Description
		486	386	286	
63 /*r*	ARPL *r/m16,r16*	9/9	*pm*=20/21	*pm*=10/11	Adjust RPL of *r/m*16 to not less than RPL of r16

The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level" --bottom two bits) of the first operand is less than the RPL field of the second operand, the zero flag is set to 1 and the RPL field of the first operand is increased to match the second operand. Otherwise, the zero flag is set to 0 and no change is made to the first operand.

ARPL appears in operating system software, not in application programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller is allowed. The second operand of ARPL is normally a register that contains the CS selector value of the caller.

BOUND

Check array index against bounds
80186 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
62 /r	BOUND r16, 7	7	10	13	Check if r16 is within m16&16 bounds (passes test)
62 /r	BOUND r32, 7	7	10		Check if r32 is within m32&32 bounds (passes test)

BOUND ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word for an operand-size attribute of 16 bits and a doubleword for an operand-size attribute of 32 bits. The first operand (a register) must be greater than or equal to the first bound in memory (lower bound), and less than or equal to the second bound in memory (upper bound). If the register is not within bounds, an Interrupt 5 occurs; the return EIP points to the BOUND instruction.

The bounds limit data structure is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array.

BSF

Bit scan forward
386 processors and greater

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks 486	Clocks 386	Description
0F BC	BSF r16,r/m16	6-42/7-43	10+3n	Bit scan forward on r/m word
0F BC	BSF r32,r/m32		10+3n	Bit scan forward on r/m dword

BSF scans the bits in the second word or doubleword operand starting with bit 0. The ZF flag is cleared if the bits are all 0; otherwise, the ZF flag is set and the destination register is loaded with the bit index of the first set bit.

BSR

Bit scan reverse
386 processors and greater

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks 486	Clocks 386	Description
0F BD	BSR r16,r/m16	6-103/7-104	10+3n	Bit scan reverse on r/m word
0F BD	BSR r32,r/m32	6-103/7-104	10+3n	Bit scan reverse on r/m dword

BSR scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is cleared if the bits are all 0; otherwise, ZF is set and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

BSWAP

Byte Swap
i486 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clock	Description
		486	
0F C8/r	BSWAP r32	1	Swap bytes to convert little/big endian data in a 32-bit register to big/little endian form.

BSWAP reverses the byte order of a 32-bit register, converting a value in little/big endian form to big/little endian form. When **BSWAP** is used with a 16-bit operand size, the result left in the destination register is undefined.

BT

Bit test
386 processors and greater

O	D	I	T	S	Z	A	P	C
								*

Opcode	Instruction	Clocks		Description
		486	386	
0F A3	BT r/m16,r16	3/8	3/12	Save bit in carry flag
0F A3	BT r/m32,r32	3/8	3/12	Save bit in carry flag
0F BA /4 ib	BT r/m16,imm8	3/3	3/6	Save bit in carry flag
0F BA /4 ib	BT r/m32,imm8	3/3	3/6	Save bit in carry flag

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag.

BTC

Bit test and complement
386 processors and greater

O	D	I	T	S	Z	A	P	C
								*

Opcode	Instruction	Clocks		Description
		486	386	
0F BB	BTC r/m16,r16	6/13	6/13	Save bit in carry flag and complement
0F BB	BTC r/m32,r32	6/13	6/13	Save bit in carry flag and complement
0F BA /7 ib	BTC r/m16,imm8	6/8	6/8	Save bit in carry flag and complement
0F BA /7 ib	BTC r/m32,imm8	6/8	6/8	Save bit in carry flag and complement

BTC saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then complements the bit.

BTR

Bit test and reset
386 processors and greater

O	D	I	T	S	Z	A	P	C
								*

Opcode	Instruction	Clocks		Description
		486	386	
0F B3	BTR r/m16,r16	6/13	6/13	Save bit in carry flag and reset
0F B3	BTR r/m32,r32	6/13	6/13	Save bit in carry flag and reset
0F BA /6 ib	BTR r/m16,imm8	6/8	6/8	Save bit in carry flag and reset
0F BA /6 ib	BTR r/m32,imm8	6/8	6/8	Save bit in carry flag and reset

BTR saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 0 in the bit.

BTS

Bit test and set
386 processors and greater

O	D	I	T	S	Z	A	P	C
								*

Opcode	Instruction	Clocks		Description
		486	386	
0F AB	BTS r/m16,r16	6/13	6/13	Save bit in carry flag and set
0F AB	BTS r/m32,r32	6/13	6/13	Save bit in carry flag and set
0F BA /5 ib	BTS r/m16,imm8	6/8	6/8	Save bit in carry flag and set
0F BA /5 ib	BTS r/m32,imm8	6/8	6/8	Save bit in carry flag and set

BTS saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 1 in the bit.

CALL

Call Procedure

O D I T S Z A P C

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Opcode	Instruction	Clocks				Description
		486	386	286*	86	
E8 cw	CALL rel16	3	7+m	7	19	Call near, displacement relative to next instruction
FF /2	CALL r/m16	5/5	7+m/10+m	7/11	16/21+EA	Call near, register indirect/memory indirect

Opcode	Instruction	Clocks 486	386	286*	86	Description
9A cd	CALL ptr16:16	18,pm=20	17+m,pm=34=m	13,pm=26	28	Call intersegment, to full pointer given
9A cd	CALL ptr16:16	pm=35	pm=52+m	41		Call gate, same privilege
9A cd	CALL ptr16:16	pm=69	pm=86+m	82		Call gate, more privilege, no parameters
9A cd	CALL ptr16:16	pm=77+4x	pm=94+4x+m	86+4x		Call gate, more privilege, x parameters
9A cd	CALL ptr16:16	pm=37+ts	ts	177/182		Call to task (via task state segment/task gate for 286
FF /3	CALL m16:16	17,pm=20	22+m,pm38+m	16/29	37+EA	Call intersegment, address at r/m dword
FF /3	CALL m16:16	pm=35	pm=56+m	44		Call gate, same privilege
FF /3	CALL m16:16	pm=69	pm=90+m	83		Call gate, more privilege, no parameters
FF /3	CALL m16:16	pm=77+4x	pm=98+4x+m	90+4x+m		Call gate, more privilege, x parameters
FF /3	CALL m16:16	pm=37+ts	5 + ts	180/185		Call to task (via task state segment/task gate for 286)
E8 cd	CALL rel32	3	7+m			Call near, displacement relative to next instruction
FF /2	CALL r/m32	5/5	7+m/10+m			Call near, indirect
9A cp	CALL ptr16:32	18,pm=20	17+m,pm=34+m			Call intersegment, to full pointer given
9A cp	CALL ptr16:32	pm=35	pm=52+m			Call gate, same privilege
9A cp	CALL ptr16:32	pm=69	pm=86+m			Call gate, more privilege, no parameters
9A cp	CALL ptr32:32	pm=77+4x	pm=94+4x+m			Call gate, more privilege, x parameters
9A cp	CALL ptr16:32	pm=37+ts	ts			Call to task
FF /3	CALL m16:32	17,pm=20	22+m,pm=38+m			Call intersegment, address at r/m dword
FF /3	CALL m16:32	pm=35	pm=56+m			Call gate, same privilege
FF /3	CALL m16:32	pm=69	pm=90+m			Call gate, more privilege, no parameters
FF /3	CALL m16:32	pm=77+4x	pm=98+4x+m			Call gate, more privilege, x parameters
FF /3	CALL m16:32	pm=37+ts	5 + ts			Call to task

*Add one clock for each byte in the next instruction executed (80286 only).

The CALL instruction causes the procedure named in the operand to be executed. When the procedure is complete (a return instruction is executed within the procedure), execution continues at the instruction that follows the CALL instruction.

The action of the different forms of the instruction are described next.

Near calls are those with destinations of type r/m16, r/m32, rel16, rel32; changing or saving the segment register value is not necessary. The CALL rel16 and CALL rel32 forms add a signed offset to the address of the instruction following CALL to determine the destination. The rel16 form is used when the instruction's operand-size attribute is 16 bits; rel32 is used when the operand-size attribute is 32 bits. The result is stored in the 32-bit EIP register. With rel 16, the upper 16 bits of EIP are cleared, resulting in an offset whose value does not exceed 16 bits. CALL r/m16 and CALL r/m32 specify a register or memory location from which the absolute segment offset is fetched. The offset fetched from r/m is 32 bits for an operand-size attribute of 32 (r/m32), or 16 bits for an operand-size of 16 (r/m16). The offset of the instruction following CALL is pushed onto the stack. It will be popped by a near RET instruction within the procedure. The CS register is not changed by this form of CALL.

The far calls, CALL ptr16:16 and CALL ptr16:32, use a 4-byte or 6-byte operand as a long pointer to the procedure called. The CALL m16:16 and m16:32 forms fetch the long pointer from the memory location specified (indirection). In real address mode or virtual 8086 mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). These forms of the instruction push both CS and IP or EIP as a return address.

In protected mode, both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers:

- a far call to the same protection level
- an inter-protection level far call
- a task switch

Note: Turbo Assembler extends the syntax of the CALL instruction to facilitate parameter passing to high-level language routines. See Chapter 7 of the Turbo Assembler User's Guide for more details.

CBW Convert byte to word

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
98	CBW	3	3	2	2	AX sign-extend of AL

CBW converts the signed byte in AL to a signed word in AX by extending the most significant bit of AL (the sign bit) into all of the bits of AH.

CDQ

Convert doubleword to quadword
386 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks		Description
		486	386	
99	CDQ	3	2	EDX:EAX [(sign-extend of EAX)

CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX.

CLC

Clear carry flag

O	D	I	T	S	Z	A	P	C
								0

Opcode	Instruction	Clocks			
		486	386	286	86
F8	CLC	2	2	2	2

CLC sets the carry flag to zero. It does not affect other flags or registers.

CLD

Clear direction flag

O	D	I	T	S	Z	A	P	C
	0							

Opcode	Instruction	Clocks				Description
		486	386	286	86	
C	CLD	2	2	2	2	Clear direction flag

CLD clears the direction flag. No other flags or registers are affected. After CLD is executed, string operations will increment the index registers (SI or DI) that they use.

CLI

Clear interrupt flag

O	D	I	T	S	Z	A	P	C
		0						

Opcode	Instruction	Clocks			
		486	386	286	86
FA	CLI	5	3	3	2

CLI clears the interrupt flag if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction or from that point on until the interrupt flag is set.

CLTS

Clear task switched flag
80286 and greater protected mode only

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

TS = 0 (TS is in CR0, not the flag register)
```

Opcode	Instruction	Clocks		
		486	386	286
0F 06	CLTS	7	5	2

CLTS clears the task-switched (TS) flag in register CR0. This flag is set by the 386 every time a task switch occurs. The TS flag is used to manage processor extensions as follows:

- Every execution of an ESC instruction is trapped if the TS flag if set.
- Execution of a WAIT instruction is trapped if the MP flag and the TS flag are both set.

Thus, if a task switch was made after an ESC instruction was begun, the processor extension's context may need to be saved before a new ESC instruction can be issued. The fault handler saves the context and resets the TS flag.

CLTS appears in operating system software, not in application programs. It is a privileged instruction that can only be executed at privilege level 0.

CMC

Complement carry flag

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F5	CMC	2	2	2	2	Complement carry flag

CMC reverses the setting of the carry flag. No other flags are affected.

CMP — Compare two operands

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
3C ib	CMP AL,imm8	1	2	3	4	Compare immediate byte to AL
3D iw	CMP AX,imm16	1	2	3	4	Compare immediate word from AX
3D id	CMP EAX,imm32	1	2			Compare immediate dword to EAX
80 /7 ib	CMP r/m8,imm8	1/2	2/5	3/6	4/10+EA	Compare immediate byte to r/m byte
81 /7 iw	CMP r/m16,imm16	1/2	2/5	3/6	4/10+EA	Compare immediate word to r/m word
81 /7 id	CMP r/m32,imm32	1/2	2/5			Compare immediate dword to r/m dword
83 /7 ib	CMP r/m16,imm8	1/2	2/5	3/6	4/10+EA	Compare sign extended immediate byte to r/m word
83 /7 ib	CMP r/m32,imm8	1/2	2/5			Compare sign extended immediate byte to r/m dword
38 /r	CMP r/m8,r8	1/2	2/5	2/7	3/9+EA	Compare byte register to r/m byte
39 /r	CMP r/m16,r16	1/2	2/5	2/7	3/9+EA	Compare word register to r/m word
39 /r	CMP r/m32,r32	1/2	2/5			Compare dword register to r/m dword
3A /r	CMP r8,r/m8	1/2	2/6	2/6	3/9+EA	Compare r/m byte to byte register
3B /r	CMP r16,r/m8	1/2	2/6	2/6	3/9+EA	Compare r/m word to word register
3B /r	CMP r32,r/m32	1/2	2/6			Compare r/m dword to dword register

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed. CMP is typically used in conjunction with conditional jumps and the SETcc instruction. If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

CMPS CMPSB CMPSW CMPSD — Compare string operands

CMPSD 386 processors and greater

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
A6	CMPS m8,m8	8	10	8	22	Compare bytes ES:[(E)DI] (second operand) with [(E)SI} (first operand)
A7	CMPS m16,m16	8	10	8	22	Compare words ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A7	CMPSm32,m32	8	10			Compare dwords ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A6	CMPSB	8	10	8	22	Compare bytes ES:[(E)DI] with DS:[SI]
A7	CMPSW	8	10	8	22	Compare words ES:[(E)DI] with DS:[SI]
A7	CMPSD	8	10			Compare dwords ES:[(E)DI] with DS:[SI]

CMPS compares the byte, word, or doubleword pointed to by the source-index register with the byte, word, or doubleword pointed to by the destination-index register.

If the address-size attribute of this instruction is 16 bits, SI and DI will be used for source- and destination-index registers; otherwise ESI and EDI will be used. Load the correct index values into SI and DI (or ESI and EDI) before executing CMPS.

The comparison is done by subtracting the operand indexed by the destination-index register from the operand indexed by the source-index register.

Note that the direction of subtraction for CMPS is [SI] - [DI] or [ESI] - [EDI]. The left operand (SI or ESI) is the source and the right operand (DI or EDI) is the destination. This is the reverse of the usual Intel convention in which the left operand is the destination and the right operand is the source.

The result of the subtraction is not stored; only the flags reflect the change. The types of the operands determine whether bytes, words, or doublewords are compared. For the first operand (SI or ESI), the DS register is used, unless a segment override byte is present. The second operand (DI or EDI) must be addressable from the ES register; no segment override is possible.

After the comparison is made, both the source-index register and destination-index register are automatically advanced. If the direction flag is 0 (CLD was executed), the registers increment; if the direction flag is 1 (STD was executed), the registers decrement. The registers increment or decrement by 1 if a byte is compared, by 2 if a word is compared, or by 4 if a doubleword is compared.

CMPSB, CMPSW and CMPSD are synonyms for the byte, word, and doubleword CMPS instructions, respectively.

CMPS can be preceded by the REPE or REPNE prefix for block comparison of CX or ECX bytes, words, or doublewords. Refer to the description of the REP instruction for more information on this operation.

CMPXCHG Compare and Exchange i486 processors and greater

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

Opcode	Instruction	Clock 486	Description
0F B0/r	CMPXCHG r/m8,r8	6/7 if comparison is successful; 6/10 if comparison fails	Compare AL with r/m byte. If equal, set ZF and load byte reg into r/m byte. Else, clear ZF and load r/m byte into AL.
0F B1/r	CMPXCHG r/m16,r16	6/7 if comparison is successful; 6/10 if comparison fails	Compare AX with r/m word. If equal, set ZF and load word reg into r/m word. Else, clear ZF and load r/m word into AX.

Opcode	Instruction	Clock	Description
		486	
0F B1/r	CMPXCHG r/m32,r32	6/7 if comparison is successful; 6/10 if comparison fails	Compare EAX with r/m dword. If equal, set ZF and load dword reg into r/m dword. Else, clear ZF and load r/m dword into EAX.
Note: The A-stepping of the i486 used the opcodes 0F A6 and 0F A7.			

The CMPXCHG instruction compares the accumulator (AL, AX, or EAX register) with DEST. If they are equal, SRC is loaded into DEST. Otherwise, DEST is loaded into the accumulator.

DEST is the destination operand; SRC is the source operand.

Protected mode exceptions: #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault code) for a page fault; #AC for an unaligned memory reference if the current privilege level is 3.

Real mode exception: interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFh.

Virtual 8086 mode exceptions: interrupt 13, as in real mode; #PF and #AC, as in protected mode.

Note: This instruction can be used with a LOCK prefix. In order to simplify interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. DEST is written back if the comparison fails, and SRC is written into the destination otherwise. (The processor never produces a locked read without producing a locked write.)

CHPXCHG8B Compare and Exchange 8 bytes

Pentium processors and greater

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks	Description
		Pentium	
0F C7	CMPXCHG64 r/m64	10	compare EDX:EAX with r/m qword. If equal, set ZF and load ECX:EBX into r/m qword. Else, clear ZF and load r/m into EDX:EAX.

The CMPXCHG8B instruction compares the 64-bit value in EDX:EAX with DEST. EDX contains the high-order 32 bits and EAX contains the low-order 32 bits of the 64-bit value. If they are equal, the 64-bit value in ECX:EBX is stored into DEST. ECX contains the high-order 32 bits and EBX contains the low-order 32 bits. Otherwise, DEST is loaded into EDX:EAX.

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected mode excpetions: #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault code) for a page fault; #AC for unaligned memory reference if the current privlege level is 3.

The destination operand must be a memory operand, not a register. If the CMPXCHG8B instruction is executed with a modr/m byte representing a register as the destination operand, #UD occurs.

Real mode exception: interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFh.

Virtual 8086 mode exceptions: same exceptions as in real mode, plus #PF(fault code) for a page fault; #AC for unalligned memory reference if the current privilage level is 3. #UD if the modr/m byte represents a register as the destination.

Notes: this instruction can be used with a LOCK prefix. In order to simplify interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. DEST is written back if the comparison fails, and SRC is written into the destination otherwise. (The processor never produces a locked read without also producing a locked write.)

The "r/m64" syntax had previously been used only in the context of floating point operations. It indicates a 64-bit value, in memory at an address determined by the modr/m byte.

CPUID

CPU identification
Pentium processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks Pentium	Description
0F A2	CPUID	14	EAX <- CP0U identification info.

The CPUID instruction provides information to software about the vendor, family, model, and stepping of microprocessor on which it is executing. An input value loaded into the EAX register for this instruction indicates what information should be returned by the CPUID instruction.

Following execution of the CPUID instruction with a zero in EAX, the EAX register contains the highest input value understood by the CPUID instruction. For the Pentium processor, the value in EAX will be one.

Also returned is a vender identification string contained in the EBX, EDX, and ECX registers. EBX contains the first four characters. For Intel processors, the vender identification string is "GenuineIntel" as follors:

EBX -- 756e6547h (* "Genu", with 'G' in the low nibble of BL *)
EDX -- 49656e69h (* "ineI", with 'i' in the low nibble of DL *)
ECX -- 6c65746eh (* "ntel", with 'n' in the low nibble of CL *)

Following execution of the CPUID instruction with an input value of one loaded into the EAX register, bits 0-3 in EAX contain the stepping id of the microprocessor, bits 4-7 of EAX contain the model (the first model will be indicated by a 0001b in these bits) and bits 8-11 of EAX contain the family (5 for the Pentium processor family). Bits 12-31 of EAX are reserved, as well as EBX, and ECX. The Pentium processor sets the feature register, EDX, to 1bfh, indicating which features the Pentium processor supports. A feature flag set to one indicates that the corresponding feature is supported. The feature set is defined as follows:

EDX (bit 0)	FPU on chip
EDX (bits 1-6)	Non-essential, proprietary information (contact Intel for more information)
EDX (bit 7)	Machine Check Exception
EDX (bit 8)	CMPXCHG8B Instruction
EDX (bits 9-31)	Reserved

Software should determine the vender identification in order to properly interpret the feature register flag bits.

This function does not affect the CPU flags.

CWD

Convert word to doubleword
386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks 486	386	286	86	Description
99	CWD	3	2	2	5	DX:AX ← sign-extend of AX

CWD converts the signed word in AX to a signed doubleword in DX:AX by extending the most significant bit of AX into all the bits of DX. Note that CWD is different from CWDE. CWDE uses EAX as a destination, instead of DX:AX.

CWDE

Convert word to doubleword
386 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
98	CWDE	3	3			EAX ← sign-extend of AX

CWDE converts the signed word in AX to a doubleword in EAX by ex-tend-ing the most significant bit of AX into the two most significant bytes of EAX. Note that CWDE is different from CWD. CWD uses DX:AX rather than EAX as a destination.

DAA

Decimal adjust AL after addition

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
27	DAA	2	4	3	4	Decimal adjust AL after addition

Execute DAA only after executing an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. The DAA instruction adjusts AL to contain the correct two-digit packed decimal result.

DAS

Decimal adjust AL after subtraction

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
2F	DAS	2	4	3	4	Decimal adjust AL after subtraction

Execute DAS only after a subtraction instruction that leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. DAS adjusts AL to contain the correct packed two-digit decimal result.

DEC

Decrement by 1

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
FE /1	DEC r/m8	1/3	2/6	2/7	3/15+EA	Decrement r/m byte by 1
FF /1	DEC r/m16	1/3	2/6	2/7	3/15+EA	Decrement r/m word by 1
	DEC r/m32	1/3	2/6			Decrement r/m dword by 1
48+rw	DEC r16	1	2	2	3	Decrement word register by 1
48+rw	DEC r32	1	2			Decrement dword register by 1

DEC subtracts 1 from the operand. DEC does not change the carry flag. To affect the carry flag, use the SUB instruction with an immediate operand of 1.

DIV

Unsigned divide

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F6 /6	DIV r/m8	16/16	14/17	14/17	80/86+EA	Unsigned divide AX by r/m byte (AL=QUO, AH=REM)
F7 /6	DIV r/m16	24/24	22/25	22/25	144/154+EA	Unsigned divide DX:AX by r/m word (AX=QUO, DX=REM)
F7 /6	DIV r/m32	40/40	38/41			Unsigned divide EDX:EAX by r/m dword (EAX=QUO, EDX=REM)

DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows:

Size	**Dividend**	**Divisor**	**Quotient**	**Remainder**
byte	AX	r/m8	AL	AH
word	DX:AX	r/m16	AX	DX
dword	EDX:EAX	r/m32	EAX	EDX (386 only)

ENTER

Make stack frame for procedure parameters 80186 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
C8 iw 00	Enter imm16,0	14	10	11	Make procedure stack frame

Opcode	Instruction	Clocks			Description
		486	386	286	
C8 iw 01	Enter imm16,1	17	12	15	Make stack frame for procedure parameters
C8 iw ib	Enter imm16,imm8	17+3n	15+4(n-1)	12+4(n-1)	Make stack frame for procedure parameters

ENTER creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame. BP (or EBP, if the operand-size attribute is 32 bits) is the current stack frame pointer.

If the operand-size attribute is 16 bits, the processor uses BP as the frame pointer and SP as the stack pointer. If the operand-size attribute is 32 bits, the processor uses EBP for the frame pointer and ESP for the stack pointer.

If the second operand is 0, ENTER pushes the frame pointer (BP or EBP) onto the stack; ENTER then subtracts the first operand from the stack pointer and sets the frame pointer to the current stack-pointer value.

For example, a procedure with 12 bytes of local variables would have an ENTER 12,0 instruction at its entry point and a LEAVE instruction before every RET. The 12 local bytes would be addressed as negative offsets from the frame pointer.

HLT Halt

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F4	HLT	4	5	2	2	Halt

HLT stops instruction execution and places the x86 in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after HLT, the saved CS:IP (or CS:EIP on an 386) value points to the instruction following HLT.

IDIV Signed divide

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F6 /7	IDIV r/m8	19/20	19	17/20	101-112/107-118+EA	Signed divide AX by r/m byte (AL=QUO, AH=REM)
F7 /7	IDIV r/m16	27/28	27	25/28	165-184/171-190+EA	Signed divide DX:AX by EA word (AX=QUO, DX=REM)
F7 /7	IDIV r/m32	43/44	43			Signed divide EDX:EAX by DWORD byte (EAX=QUO, EDX=REM)

IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit r/m operand. The type of the divisor determines which registers to use as follows:

Size	Divisor	Quotient	Remainder	Dividend
byte	r/m8	AL	AH	AX
word	r/m16	AX	DX	DX:AX
dword	r/m32	EAX	EDX	EDX:EAX (386 only)

If the resulting quotient is too large to fit in the destination, or if the division is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend and the absolute value of the remainder is always less than the absolute value of the divisor.

IMUL Signed multiply

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F6 /5	IMUL r/m8	13-18/13-18	9-14/12-17	13/16	80-98/86-104 +EA	AX ←AL * r/m byte
F7 /5	IMUL r/m16	13-26/13-26	9-22/12-25	21/24	128-154/134-160+EA	DX:AX ←AX * r/m word
F7 /5	IMUL r/m32	12-42/13-42	9-38/12-41			EDX:EAX ←EAX* r/m dword
0F AF /r	IMUL r16,r/m16	13-26/13-26	9-22/12-25			word register ←word register * r/m word
0F AF /r	IMUL r32,r/m32	13-42/13-42	9-38/12-41			dword register ←dword register * r/m dword
6B /r ib	IMUL r16,r/m16,imm8	13-26/13-26	9-14/12-17	21/24		word register ←r/m16 * sign-extended immediate byte
6B /r ib	IMUL r32,r/m32,imm8	13-42	9-14/12-17			dword register ←r/m32 * sign-extended immediate byte
6B /r ib	IMUL r16,imm8	13-26	9-14/12-17	21/24		word register ←word register * sign-extended immediate byte

Opcode	Instruction	Clocks				Description
		486	386	286	86	
6B /r ib	IMUL r32,imm8	13-42	9-14/12-17			dword register ←dword register * sign-extended immediate byte
69 /r iw	IMUL r16,r/ m16,imm16	13-26/13-26	9-22/12-25	21/24		word register ←r/m16 immediate word
69 /r id	IMUL r32,r/ m32,imm32	13-42/13-42	9-38/12-41			dword register r/m32 * immediate dword
69 /r iw	IMUL r16,imm16	13-26/13-26	9-22/12-25			word register ←r/m16 * immediate word
69 /r id	IMUL r32,imm32	13-42/13-42	9-38/12-41			dword register ←r/m32 * immediate dword

IMUL performs signed multiplication. Some forms of the instruction use implicit register operands. The operand combinations for all forms of the instruction are shown in the "Description" column above.

IMUL clears the overflow and carry flags under the following conditions:

Instruction form	**Condition for clearing CF and OF**
r/m8	AL = sign-extend of AL to 16 bits
r/m16	AX = sign-extend of AX to 32 bits
r/m32	EDX:EAX = sign-extend of EAX to 32 bits
r16,r/m16	Result exactly fits within r16
r32,r/m32	Result exactly fits within r32
r16,r/m16,imm16	Result exactly fits within r16
r32,r/m32,imm32	Result exactly fits within r32

IN Input from port

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

Opcode	Instruction	Clocks				
		486	386	286	86	
E4 ib	IN AL,imm8	14,pm=8*/28**,vm=27	12,pm=6*/26**	5	10	Input byte from immediate port into AL
E5 ib	IN AX,imm8	14,pm=8*/28**,vm=27	12,pm=6*/26**	5	10	Input word from immediate port into AX
E5 ib	IN EAX,imm8	14,pm=8*/28**,vm=27	12,pm=6*/26**			Input dword from immediate port into EAX
EC	IN AL,DX	14,pm=8*/28**,vm=27	13,pm=7*/27**	5	8	Input byte from port DX into AL
ED	IN AX,DX	14,pm=8*/28**,vm=27	13,pm=7*/27**	5	8	Input word from port DX into AX
ED	IN EAX,DX	14,pm=8*/28**,vm=27	13,pm=7*/27**			Input dword from port DX into EAX

*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the

DX register and using an IN instruction with DX as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

INC — Increment by 1

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
FE /0	INC r/m8	1/3	2/6	2/7	3/15+EA	Increment r/m byte by 1
FF /0	INC r/m16	1/3	2/6	2/7	3/15+EA	Increment r/m word by 1
FF /6	INC r/m32	1/3				Increment r/m dword by 1
40+ rw	INC r16	1	2	2	3	Increment word register by 1
40+ rd	INC r32	1				Increment dword register by 1

INC adds 1 to the operand. It does not change the carry flag. To affect the carry flag, use the ADD instruction with a second operand of 1.

INS INSB INSW INSD — Input from port to string 80186 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
6C	INS r/m8,DX	17,pm=10*/32**,vm=30	15,pm=9*/29**	5	Input byte from port DX into ES:(E)DI
6D	INS r/m16,DX	17,pm=10*/32**,vm=30	15,pm=9*/29**	5	Input word from port DX into ES:(E)DI
6D	INS r/m32,DX	17,pm=10*/32**,vm=30	15,pm=9*/29**		Input dword from port DX into ES:(E)DI
6C	INSB	17,pm=10*/32**,vm=30	15,pm=9*/29**	5	Input byte from port DX into ES:(E)DI
6D	INSW	17,pm=10*/32**,vm=30	15,pm=9*/29**	5	Input word from port DX into ES:(E)DI
6D	INSD	17,pm=10*/32**,vm=30	15,pm=9*/29**		Input dword from port DX into ES:(E)DI

*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

INS transfers data from the input port numbered by the DX register to the memory byte or word at ES:dest-index. The memory operand must be addressable from ES; no segment override is possible. The destination register is DI if the address-size attribute of the instruction is 16 bits, or EDI if the address-size attribute is 32 bits.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the INS instruction.

The destination address is determined by the contents of the destination index register. Load the correct index into the destination index register before executing INS.

After the transfer is made, DI or EDI advances automatically. If the direction flag is 0 (CLD was executed), DI or EDI increments; if the direction flag is 1 (STD was executed), DI or EDI decrements. DI increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a doubleword is input.

INSB, INSW and INSD are synonyms of the byte, word, and doubleword INS instructions. INS can be preceded by the REP prefix for block input of CX bytes or words. Refer to the REP instruction for details of this operation.

INT
INTO

Call to interrupt procedure

O	D	I	T	S	Z	A	P	C
		0	0					

Opcode	Instruction	Clocks				Description
		486	386	286	86	
CC	INT3	26	33	23	52	Interrupt 3--trap to debugger
CC	INT3	44	pm=59	40		Interrupt 3--protected mode
CC	INT3	71	pm=99	78		Interrupt 3--protected mode
CC	INT3	82	pm=119			Interrupt 3--from V86 mode to PL0
CC	INT3	37+ts	ts	167		Interrupt 3--protected mode
CD ib	INTimm8	30	37	23	51	Interrupt numbered by immediate byte
CD ib	INTimm8	44	pm=59	40		Interrupt--protected mode
CD ib	INTimm8	77	pm=99	78		Interrupt--protected mode
CD ib	INTimm8	86	pm=119			Interrupt--from V86 mode to PL0
CD ib	INTimm8	37+ts	ts	167		Interrupt--protected mode
CE	INTO	Pass:28, Fail:3	Fail:3, pm=3; Pass:35	Fail:3, Pass:24	Fail:4, Pass:53	Interrupt 4--if overflow flag is 1
CE	INTO	46	pm=59	41		Interrupt 4--Protected mode
CE	INTO	73	pm=99	79		Interrupt 4--Protected mode
CE	INTO	84	pm=119			Interrupt 4--from V86 mode to PL0
CE	INTO	39+ts	ts	168		Interrupt 4--Protected mode

* Add one clock for each byte of the next instruction executed (80286 only).

The INT n instruction generates via software a call to an interrupt handler. The immediate operand, from 0 to 255, gives the index number into the interrupt descriptor table (IDT) of the interrupt routine to be called. In protected mode, the IDT consists of an array of eight-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. In real address mode, the IDT is an array of four byte-long pointers. In protected and real address modes, the base linear address of the IDT is defined by the contents of the IDTR.

The INTO conditional software instruction is identical to the INT n interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made if the 86, 286, or 386 overflow flag is set.

The first 32 interrupts are reserved by Intel for system use. Some of these interrupts are use for internally generated exceptions.

INT n generally behaves like a far call except that the flags register is pushed onto the stack before the return address. Interrupt procedures return via the IRET instruction, which pops the flags and return address from the stack.

In real address mode, INT n pushes the flags, CS and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

INVD

Invalidate cache
i486 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clock	Description
		486	
0F 08	INVD	4	Invalidate entire cache

The internal cache is flushed, and a special-function bus cycle is issued which indicates that external caches should also be flushed. Data held in write-back external caches is discarded.

Note: This instruction is implementation-dependent; its function might be implemented differently on future Intel processors.

It is the responsibility of hardware to respond to the external cache flush indication.

INVLPG

Invalidate TLB entry
i486 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clock	Description
		486	
0F 01/7	INVLPG m	12 for hit	Invalidate TLB entry

The INVLPG instruction is used to invalidate a single entry in the TLB, the cache used for table entries. If the TLB contains a valid entry that maps the address of the memory operand, that TLB entry is marked invalid.

In both protected mode and virtual 8086 mode, an invalid opcode is generated when used with a register operand.

Note: This instruction is implementation-dependent; its function might be implemented differently on future Intel processors.

IRET IRETD IRETW

Interrupt return

IRETD 386 processors and greater

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

The flags register is popped from stack.

Opcode	Instruction	Clocks				Description
		486	386	286	86	
CF	IRETW	15	22,pm=38	17,pm=31	32	Interrupt return (far return and pop flags)
CF	IRETW	36	pm=82	55		Interrupt return
CF	IRETW	ts+32	ts	169		Interrupt return
CF	IRETD	15	22,pm=38			Interrupt return (far return and pop flags)
CF	IRETD	36	pm=82			Interrupt return to lesser privilege
CF	IRETD	15	pm=60			Interrupt return to V86 mode
CF	IRETD	ts+32	ts			Interrupt return
CF	IRET					Selects IRETW or IRETD depending on segment size of 16 or 32 bits. Only works for VERSION T320 or higher.

* Add one clock, for each byte in the next instruction executed (80286 only).

In real address mode, IRET pops the instruction pointer, CS, and the flags register from the stack and resumes the interrupted routine.

In protected mode, the action of IRET depends on the setting of the nested task flag (NT) bit in the flag register. When popping the new flag image from the stack, the IOPL bits in the flag register are changed only when CPL equals 0.

If NT equals 0, IRET returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, IRET also pops the stack pointer and SS from the stack.

If NT equals 1, IRET reverses the operation of a CALL or INT that caused a task switch. The updated state of the task executing IRET is saved in its task state segment. If the task is re-entered later, the code that follows IRET is executed.

IRETW pops WORD-style (if you use VERSION T320 or higher). If you're using VERSION T310 or less, use IRET; IRETW replaces old functionality of IRET.

Jcc Jump if condition is met

O D I T S Z A P C

Opcode	Instruction	Clocks 486	386	286	86	Description
77 cb	JA rel8	3/1	7+m,3	7,3	16,4	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	3/1	7+m+,3	7,3	16,4	Jump short if above or equal (CF=0)
72 cb	JB rel8	3/1	7+m,3	7,3	16,4	Jump short if below (CF=1)
76 cb	JBE rel8	3/1	7+m,3	7,3	16,4	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	3/1	7+m,3	7,3	16,4	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	3/1	9+m,5	8,4	18,6	Jump short if CX register is 0
E3 cb	JECXZ rel8	3/1	9+m,5			Jump short if ECX register is 0
74 cb	JE rel8	3/1	7+m,3	7,3	16,4	Jump short if equal (ZF=1)
74 cb	JZ rel8	3/1	7+m,3	7,3	16,4	Jump short if 0 (ZF=1)
7F cb	JG rel8	3/1	7+m,3	7,3	16,4	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	3/1	7+m,3	7,3	16,4	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	3/1	7+m,3	7,3	16,4	Jump short if less (SF<>OF)
7E cb	JLE rel8	3/1	7+m,3	7,3	16,4	Jump short if less or equal (ZF=1 and SF<>OF)
76 cb	JNA rel8	3/1	7+m,3	7,3	16,4	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	3/1	7+m,3	7,3	16,4	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	3/1	7+m,3	7,3	16,4	Jump short if not below (CF=0)
77 cb	JNBE rel8	3/1	7+m,3	7,3	16,4	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	3/1	7+m,3	7,3	16,4	Jump short if not carry (CF=0)
75 cb	JNE rel8	3/1	7+m,3	7,3	16,4	Jump short if not equal (ZF=0)
7E cb	JNG rel8	3/1	7+m,3	7,3	16,4	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	3/1	7+m,3	7,3	16,4	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	3/1	7+m,3	7,3	16,4	Jump short if not less (SF=OF)
7F cb	JNLE rel8	3/1	7+m,3	7,3	16,4	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	3/1	7+m,3	7,3	16,4	Jump short if not overflow (OF=0)
7B cb	JNP rel8	3/1	7+m,3	7,3	16,4	Jump short if not parity (PF=0)
79 cb	JNS rel8	3/1	7+m,3	7,3	16,4	Jump short if not sign (SF=0)
75 cb	JNZ rel8	3/1	7+m,3	7,3	16,4	Jump short if not zero (ZF=0)
70 cb	JO rel8	3/1	7+m,3	7,3	16,4	Jump short if overflow (OF=1)
7A cb	JP rel8	3/1	7+m,3	7,3	16,4	Jump short if parity (PF=1)
7A cb	JPE rel8	3/1	7+m,3	7,3	16,4	Jump short if parity even (PF=1)
7B cb	JPO rel8	3/1	7+m,3	7,3	16,4	Jump short if parity odd (PF=0)
78 cb	JS rel8	3/1	7+m,3	7,3	16,4	Jump short if sign (SF=1)
74 cb	JZ rel8	3/1	7+m,3	7,3	16,4	Jump short of zero (ZF=1)
0F 87 cw/cd	JA rel16/32	3/1	7+m,3			Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	3/1	7+m,3			Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	3/1	7+m,3			Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	3/1	7+m,3			Jump near if below or equal (CF=1or ZF=1)
0F 82 cw/cd	JC rel16/32	3/1	7+m,3			Jump near if carry (CF=1)
0F 84 cw/cd	JE rel16/32	3/1	7+m,3			Jump near if equal (ZF=1)

Opcode	Instruction	Clocks				Description
		486	386	286	86	
0F 84 cw/cd	JZ rel16/32	3/1	7+m,3			Jump near if 0 (ZF=1)
0F 8F cw/cd	JG rel16/32	3/1	7+m,3			Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE rel16/32	3/1	7+m,3			Jump near if greater or equal (SF=OF)
0F 8C cw/cd	JL rel16/32	3/1	7+m,3			Jump near if less (SF<>OF)
0F 8E cw/cd	JLE rel16/32	3/1	7+m,3			Jump near if less or equal(ZF=1 and SF<>OF)
0F 86cw/cd	JNA rel16/32	3/1	7+m,3			Jump near if not above (CF=1 or ZF=1)
0F 82 cw/cd	JNAE rel16/32	3/1	7+m,3			Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB rel16/32	3/1	7+m,3			Jump near if not below (CF=0)
0F 87 cw/cd	JNBE rel16/32	3/1	7+m,3			Jump near if not below or equal (CF=0 and ZF=0
0F 83 cw/cd	JNC rel16/32	3/1	7+m,3			Jump near if not carry and ZF=0)
0F 85 cw/cd	JNE rel16/32	3/1	7+m,3			Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG rel16/32	3/1	7+m,3			Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE rel16/32	3/1	7+m,3			Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL rel16/32	3/1	7+m,3			Jump near if not less (SF=OF)
0F 8F cw/cd	JNLE rel16/32	3/1	7+m,3			Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 cw/cd	JNO rel16/32	3/1	7+m,3			Jump near if not overflow (OF=0)
0F 8B cw/cd	JNP rel16/32	3/1	7+m,3			Jump near if not parity (PF=0)
0F 89 cw/cd	JNS rel16/32	3/1	7+m,3			Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ rel16/32	3/1	7+m,3			Jump near if not zero (ZF=0)
0F 80 cw/cd	JO rel16/32	3/1	7+m,3			Jump near if overflow (OF=1)
0F 8A cw/cd	JP rel16/32	3/1	7+m,3			Jump near if parity (PF=1)
0F 8A cw/cd	JPE rel16/32	3/1	7+m,3			Jump near if parity even (PF=1)
0F 8B cw/cd	JPO rel16/32	3/1	7+m,3			Jump near if parity odd (PF=0)
0F 88 cw/cd	JS rel16/32	3/1	7+m,3			Jump near if sign (SF=1)
0F 84 cw/cd	JZ rel16/32	3/1	7+m,3			Jump near if zero (ZF=1)

* When a jump is taken, add one clock for every byte of the next instruction executed (80286 only).

Note: The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). rel16/32 indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operand-size attribute of the instruction.

Conditional jumps (except JCXZ/JECXZ) test the flags which have been set by a previous instruction. The conditions for each mnemonic are given in parentheses after each description above. The terms "less" and "greater" are used for comparisons of signed integers; "above" and "below" are used for unsigned integers.

If the given condition is true, a jump is made to the location provided as the operand. Instruction coding is most efficient when the target for the conditional jump is in the current code segment and within -128 to + 127 bytes of the next instruction's first byte. The jump can also target -32768 through +32767 (segment size attribute 16) or -2 to the 31st power +2 to the 31st power -1 (segment size attribute 32) relative to the next instruction's first byte. When the target for the conditional jump is in a different segment, use the opposite case of the jump instruction (that is, JE and JNE), and then access the target with an unconditional far jump to the other segment. For example, you cannot code

```
JZ FARLABEL;
```

You must instead code

```
        JNZ  BEYOND;
        JMP  FARLABEL;
BEYOND:
```

Because there can be several ways to interpret a particular state of the flags, TASM provides more than one mnemonic for most of the conditional jump opcodes. For example, if you compared two characters in AX and want to jump if they are equal, use JE; or, if you ANDed AX with a bit field mask and only want to jump if the result is 0, use JZ, a synonym for JE.

JCXZ/JECXZ differs from other conditional jumps because it tests the contents of the CX or ECX register for 0, not the flags. JCXZ/JECXZ is useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE TARGET LABEL). The JCXZ/JECXZ prevents entering the loop with CX or ECX equal to zero, which would cause the loop to execute 64K or 32G times instead of zero times.

JMP Jump

O D I T S Z A P C

All if a task switch takes place; none if no task switch occurs

Opcode	Instruction	Clocks				Description
		486	386	286	86	
EB cb	JMP rel8	3	7+m	7	15	Jump short
E9 cw	JMP rel16	3	7+m	7	15	Jump near
FF /4	JMP r/m16	5/5	7+m/10+m	7/11	11/18+EA	Jump near indirect
EA cd	JMP ptr16:16	17pm=19	12+m, pm=27+m	11,pm=23	15	Jump intersegment, 4-byte immediate address
EA cd	JMP ptr16:16	32	pm=45+m	38		Jump to call gate, same privilege
EA cd	JMP ptr16:16	42+ts	ts	175		Jump via task state segment
EA cd	JMP ptr16:16	43+ts	ts	180	24+EA	Jump via task gate
FF /5	JMP m16:16	13,pm=18	43+m,pm=31+m	15,pm=26		Jump r/m16:16 indirect and intersegment
FF /5	JMP m16:16	31	pm=49+m	41		Jump to call gate, same privilege
FF /5	JMP m16:16	41+ts	5+ts	178		Jump via task state segment
FF /5	JMP m16:16	42+ts	5+ts	183		Jump via task gate
E9 cd	JMP rel32	3	7+m			Jump near
FF /4	JMP r/m32	5/5	7+m,10+m			Jump near
EA cp	JMP ptr16:32	13,pm=18	12+m, pm=27+m			Jump intersegment, 6-byte immediate address

Opcode	Instruction	Clocks				Description
		486	386	286	86	
EA cp	JMP ptr16:32	31	pm=45+m			Jump to call gate, same privilege
EA cp	JMP ptr16:32	42+ts	ts			Jump via task state segment
EA cp	JMP ptr16:32	43+ts	ts			Jump via task gate
FF /5	JMP m16:32	13,pm=18	43+m, pm=31+m			Jump intersegment address at r/m dword
FF /5	JMP m16:32	31	pm=49+m			Jump to call gate, same privilege
FF /5	JMP m16:32	41+ts	5 + ts			Jump via task state segment
FF /5	JMP m16:32	42+ts	5 + ts			Jump via task gate

* Add one clock for every byte of the next instruction executed (80286 only).

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

The action of the various forms of the instruction are shown below.

Jumps with destinations of type r/m16, r/m32, rel16, and rel32 are near jumps and do not involve changing the segment register value.

The JMP rel16 and JMP rel32 forms of the instruction add an offset to the address of the instruction following the JMP to determine the destination. The rel16 form is used when the instruction's operand-size attribute is 16 bits (segment size attribute 16 only); rel32 is used when the operand-size attribute is 32 bits (segment size attribute 32 only). The result is stored in the 32-bit EIP register. With rel16, the upper 16 bits of EIP are cleared, which results in an offset whose value does not exceed 16 bits.

JMP r/m16 and JMP r/m32 specifies a register or memory location from which the absolute offset from the procedure is fetched. The offset fetched from r/m is 32 bits for an operand-size attribute of 32 bits (r/m32), or 16 bits for an operand-size attribute of 16 bits (r/m16).

The JMP ptr16:16 and ptr16:32 forms of the instruction use a four-byte or six-byte operand as a long pointer to the destination. The JMP m16:16 and m16:32 forms fetch the long pointer from the memory location specified (indirection). In real address mode or virtual 8086 mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). In protected mode, both long pointer forms consult the access rights (AR) byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the jump will perform one of the following types of control transfers:

- a jump to a code segment at the same privilege level
- a task switch

LAHF

Loads flags into AH register

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
9F	LAHF	3	2	2	4	Load: AH = flags SF ZF xx AF xx PF xx CF

LAHF transfers the low byte of the flags word to AH. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary carry, indeterminate, parity, indeterminate, and carry.

LAR

Load access rights byte
80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 02/r	LAR r16,r/m16	11/11	pm=15/16	14/16	r16←r/m16 masked by FF00
0F 02 /r	LAR r32,r/m32	11/11	pm=15/16		r32←r/m32 masked by 00FxFF00

The LAR instruction stores a marked form of the second doubleword of the descriptor for the source selector if the selector is visible at the CPL (modified by the selector's RPL) and is a valid descriptor type. The destination register is loaded with the high-order doubleword of the descriptor masked by 00FxFF00, and ZF is set to 1. The x indicates that the four bits corresponding to the upper four bits of the limit are undefined in the value loaded by LAR. If the selector is invisible or of the wrong type, ZF is cleared.

If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16-bits of this value are stored in the 16-bit destination register.

All code and data segment descriptors are valid for LAR. (See your Intel manual for valid segment and gate descriptor types for LAR.)

LEA

Load effective address offset

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
8D/r	LEA r16,m	1	2	3	2+EA	Store effective address for m in register r16
8D/r	LEA r32,m	1	2			Store effective address for m in register r32

Opcode	Instruction	Clocks				Description
		486	386	286	86	
8D/r	LEA r16,m	1	2			Store effective address for m in register r16
8D/r	LEA r32,m	1	2			Store effective address for m in register r32

LEA calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction is determined by the chosen register. The address-size attribute is determined by the USE attribute of the segment containing the second operand. The address-size and operand-size attributes affect the action performed by LEA, as follows:

Operand size	**Address size**	**Action performed**
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

LEAVE

High-level procedure exit
80186 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
C9	LEAVE	5	4	5	Set SP to BP
C9	LEAVE	5	4		Set ESP to EBP

LEAVE reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, LEAVE releases the stack space used by a procedure for its local variables. The old frame pointer is popped into BP or EBP, restoring the caller's frame. A subsequent RET nn instruction removes any arguments pushed onto the stack of the exiting procedure.

LGDT/LIDT

Load global/interrupt descriptor table register
80286 and greater protected mode only

O D I T S Z A P C

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 01 /2	LGDT m16&32	11	11	11	Load m into global descriptor table register
0F 01 /3	LIDT m16&32	11	11	12	Load m into interrupt descriptor table register

The LGDT and LIDT instructions load a linear base address and limit value from a six-byte data operand in memory into the GDTR or IDTR, respectively. If a 16-bit operand is used with LGDT or LIDT, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order 8 bits of the 6-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base is loaded; the high-order 8 bits of the 6-byte operand are used as high-order base address bits.

The SGDT and SIDT instructions always store into all 48 bits of the 6-byte data operand. With the 80286, the upper 8 bits are undefined after SGDT or SIDT is executed. With the 386, the upper 8 bits are written with the high-order 8 address bits, for both a 16-bit operand and a 32-bit operand. If LGDT or LIDT is used with a 16-bit operand to load the register stored by SGDT or SIDT, the upper 8 bits are stored as zeros.

LGDT and LIDT appear in operating system software; they are not used in application programs. They are the only instructions that directly load a linear address (i.e., not a segment relative address) in 386 protected mode.

LGS LSS LFS LDS LES

Load full pointer
LGS/LSS/LFS 386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
C5 /r	LDS r16,m16:16	6/12	7,pm=22	7,pm=21	16+EA	Load DS:r16 with pointer from memory
C5 /r	LDS r32,m16:32	6/12	7,pm=22			Load DS:r32 with pointer from memory
0F B2 /r	LSS r16,m16:16	6/12	7,pm=22			Load SS:r16 with pointer from memory
0F B2 /r	LSS r32,m16:32	6/12	7,pm=22			Load SS:r32 with pointer from memory
C4 /r	LES r16,m16:16	6/12	7,pm=22	7,pm=21	16+EA	Load ES:r16 with pointer from memory
C4 /r	LES r32,m16:32	6/12	7,pm=22			Load ES:r32 with pointer from memory
0F B4 /r	LFS r16,m16:16	6/12	7,pm=25			Load FS:r16 with pointer from memory
0F B4 /r	LFS r32,m16:32	6/12	7,pm=25			Load FS:r32 with pointer from memory
0F B5 /r	LGS r16,m16:16	6/12	7,pm=25			Load GS:r16 with pointer from memory
0F B5 /r	LGS r32,m16:32	6/12	7,pm=25			Load GS:r32 with pointer from memory

These instructions read a full pointer from memory and store it in the selected segment register: register pair. The full pointer loads 16 bits into the segment register SS, DS, ES, FS, or GS. The other register loads 32 bits if the operand-size attribute is 32 bits, or loads 16 bits if the operand-size attribute is 16 bits. The other 16- or 32-bit register to be loaded is determined by the r16 or r32 register operand specified.

When an assignment is made to one of the segment registers, the de-scriptor is also loaded into the segment register. The data for the register is obtained from the descriptor table entry for the selector given.

A null selector (values 0000-0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector to address memory causes a #GP(0) exception. No memory reference to the segment occurs.)

LLDT

Load local descriptor table register
80286 and greater protected mode only

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

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 00 /2	LLDT r/m 16	11/11	20	17/19	Load selector r/m16 into LDTR

LLDT loads the local descriptor table register (LDTR). The word operand (memory or register) to LLDT should contain a selector to the global descriptor table (GDT). The GDT entry should be a local descriptor table. If so, then the LDTR is loaded from the entry. The descriptor registers DS, ES, SS, FS, GS, and CS are not affected. The LDT field in the task state segment does not change.

The selector operand can be 0; if so, the LDTR is marked invalid. All descriptor references (except by the LAR, VERR, VERW or LSL instructions) cause a #GP fault.

LLDT is used in operating system software; it is not used in application programs.

LMSW

Load machine status word
80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 01 /6	LMSW r/m 16	13/13	10/13	3/6	Load r/m 16 into machine status word

LMSW loads the machine status word (part of CR0) from the source operand. This instruction can be used to switch to protected mode; if so, it must be followed by an intrasegment jump to flush the instruction queue. LMSW will not switch back to real address mode.

LMSW is used only in operating system software. It is not used in application programs.

LOCK

Assert LOCK# signal prefix

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F0	LOCK	1	0	0	2	Assert LOCK# signal for the next instruction

The LOCK prefix causes the LOCK# signal of the CPU to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the CPU has exclusive use of any shared memory while LOCK# is asserted. The read-modify-write sequence typically used to implement test-and-set on the 386 is the BTS instruction.

On the 386 and i486, the LOCK prefix functions only with the following instructions:

BT, BTS, BTR, BTC	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR	mem, reg/imm
NOT, NEG, INC, DEC	mem

An undefined opcode trap will be generated if a LOCK prefix is used with any instruction not listed above.

XCHG always asserts LOCK # regardless of the presence or absence of the LOCK prefix.

The integrity of the LOCK is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Locked access is not assured if another CPU processor is executing an instruction concurrently that has one of the following characteristics:

- Is not preceded by a LOCK prefix.
- Is not one of the instructions in the preceding list.
- Specifies a memory operand that does not exactly overlap the destination operand. Locking is not guaranteed for partial overlap, even if one memory operand is wholly contained within another.

LODS LODSB LODSW LODSD

Load string operand

LODSD 386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
AC	LODS m18	5	5	5	12	Load byte [(E)SI] into AL
AD	LODS m16	5	5	5	12	Load word [(E)SI] into AX
AD	LODS m32	5	5			Load dword [(E)SI] into EAX
AC	LODSB	5	5	5	12	Load byte DS:[(E)SI] into AL
AD	LODSW	5	5	5	12	Load word DS:[(E)SI] into AX
AD	LODSD5	5				Load dword DS:[(E)SI] into EAX

LODS loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed to by the source-index register. After the transfer is made, the source-index register is automatically advanced. If the direction flag is 0 (CLD was executed), the source index increments; if the direction flag is 1 (STD was executed), it decrements. The increment or decrement is 1 if a byte is loaded, 2 if a word is loaded, or 4 if a doubleword is loaded.

If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise the address-size attribute is 32 bits, and the ESI register is used. The address of the source data is determined solely by the contents of ESI/SI. Load the correct index value into SI before executing the LODS instruction. LODSB, LODSW, LODSD are synonyms for the byte, word, and doubleword LODS instructions.

LODS can be preceded by the REP prefix; however, LODS is used more typically within a LOOP construct, because further processing of the data moved into EAX, AX, or AL is usually necessary.

LOOP
LOOPcond

Loop control with CX counter
Loop control with CX/ECX counter
386 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
E2 cb	LOOP rel8	2,6	11+m	8,noj=4	17,noj=5	DEC Count; jump short if Count 0
E1 cb	LOOPE rel8	9,6	11+m	8,noj=4	18,noj=6	DEC Count; jump short if Count 0 and ZF=1
E1 cb	LOOPZ rel8	9,6	11+m	8,noj=4	18,noj=6	DEC Count; jump short if Count 0 and ZF=1
E0 cb	LOOPNE rel8	9,6	11+m	8,noj=4	19,noj=5	DEC Count; jump short if Count 0 and ZF=0
E0 cb	LOOPNZ rel8	9,6	11+m	8,noj=4	19,noj=5	DEC Count; jump short if Count 0 and ZF=0

LOOP decrements the count register without changing any of the flags. Conditions are then checked for the form of LOOP being used. If the conditions are met, a short jump is made to the label given by the operand to LOOP. If the address-size attribute is 16 bits, the CX register is used as the count register; otherwise the ECX register is used (386 only). The operand of LOOP must be in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

The LOOP instructions provide iteration control and combine loop index management with conditional branching. Use the LOOP instruction by loading an unsigned iteration count into the count register, then code the LOOP at the end of a series of instructions to be iterated. The destination of LOOP is a label that points to the beginning of the iteration.

LSL

Load segment limit
80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 03 /r	LSL r16,r/m16	10/10	pm=20/21	14/16	Load: r16←segment limit, selector r/m16 (byte granular)
0F 03 /r	LSL r32,r/m32	10/10	pm=20/21		Load: r32←segment limit, segment limit, selector r/m32 (byte granular)
0F 03 /r	LSL r16,r/m16	10/10	pm=25/26	14/16	Load: r16←segment limit, segment limit, selector r/m16 (page granular)
0F 03 /r	LSL r32,r/m32	10/10	pm=26/26		Load: r32←segment limit selector r/m32 (page granular)

The LSL instruction loads a register with an unscrambled segment limit, and sets ZF to 1, provided that the source selector is visible at the CPL weakened by RPL, and that the descriptor is a type accepted by LSL. Otherwise, ZF is cleared to 0, and the destination register is unchanged. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, LSL will translate it to a byte limit before loading it in the destination register (shift left 12 the 20-bit "raw" limit from descriptor, then OR with 00000FFFH).

The 32-bit forms of this instruction store the 32-bit byte granular limit in the 16-bit destination register.

Code and data segment descriptors are valid for LSL.

LTR

Load task register
80286 and greater protected mode only

O D I T S Z A P C

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 00 /3	LTR r/m16	20/20	pm=23/27	17/19	Load EA word into task register

LTR loads the task register from the source register or memory location specified by the operand. The loaded task state segment is marked busy. A task switch does not occur.

LTR is used only in operating system software; it is not used in application programs.

MOV

Move data

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
88 /r	MOV r/m8,r8	1	2/2	2/3	2/9+EA	Move byte register into r/m byte
89 /r	MOV r/m16,r16	1	2/2	2/3	2/9+EA	Move word register into r/m word
89 /r	MOV r/m32,r32	1	2/2			Move dword register to r/m dword
8A /r	MOV r8,r/m8	1	2/4	2/5	2/8+EA	Move r/m byte into byte register
8B /r	MOV r16,r/m16	1	2/4	2/5	2/8+EA	Move r/m word into word register
8B /r	MOV r32,r/m32	1	2/4			Move r/m dword into dword register
8C /r	MOV r/m16,Sreg	3/3	2/2	2/3	2/9+EA	Move segment register to r/m register
8D /r	MOV Sreg,r/m16	3/9	2/5,pm= 1/198	2/5,pm= 17/19	2/8+EA	Move r/m word to segment register
A0	MOV AL,moffs8	1	4	5	10	Move byte at (seg:offset) to AX
A1	MOV AX,moffs16	1	4	5	10	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32	1	4			Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	1	4	3	10	Move AL to (seg:offset)
A3	MOV moffs16,AX	1	2	3	10	Move AX to (seg:offset)
A3	MOV moffs32,EAX	1	2			Move EAX to (seg:offset)
B0+ rb	MOV reg8,imm8	1	2	2	4	Move immediate byte to register
B8+ rw	MOV reg16,imm16	1	2	2	4	Move immediate word to register
B8+rd	MOV reg32,imm32	1	2			Move immediate dword to register

Opcode	Instruction	Clocks				Description
		486	386	286	86	
C6	MOV r/m8,imm8	1	2/2	2/3	4/10+EA	Move immediate byte to r/m byte
C7	MOV r/m16,imm16	1	2/2	2/3	4/10+EA	Move immediate word to r/m word
C7	MOV r/m32,imm32	1	2/2			Move immediate dword to r/m dword

MOV copies the second operand to the first operand.

If the destination operand is a segment register (DS, ES, SS, etc.), then data from a descriptor is also loaded into the register. The data for the register is obtained from the descriptor table entry for the selector given. A null selector (values 0000-0003) can be loaded into DS and ES registers without causing an exception; however, use of DS or ES causes a #GP(0), and no memory reference occurs.

A MOV into SS inhibits all interrupts until after the execution of the next instruction (which is presumably a MOV into eSP).

MOV

Move to/from special registers
386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks		Description
		486	386	
0F 22 /r	MOV,CR0,r32	16		Move (register) to (control register)
0F 20 /r	MOV r32,CR0/CR2/CR3/CR4	4	6	Move (control register) to (register)
0F 22 /r	MOV CR0/CR2/CR3/CR4,r32	4	10/4/5	
0F 21 /r	MOV r32,DR0 - 3	10	22	Move (debug register) to (register)
0F 21 /r	MOV r32,DR6/DR7	10	14	Move (debug register) to (register)
0F 23 /r	MOV DR0 -3,r32	11	22	Move (register) to (debug register)
0F 23 /r	MOV DR6/DR7,r32	11	16	Move (register) to (debug register)
0F 24 /r	MOV r32,TR6/TR7	4	12	Move (test register) to (register)
0F 26 /r	MOV TR6/TR7,r32	4	12	Move (register) to (test register)
0F 24 /r	MOV r32,TR3		3	Move (registers) to (test register3)

These forms of MOV store or load the following special registers in or from a general-purpose register:

- Control Registers CRO, CR2, CR3, and CR4 (CR4 only on Pentium)
- Debug Registers DRO, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR3, TR4, TR5, TR6, and TR7 (not valid on Pentium)

32-bit operands are always used with these instructions, regardless of the operand-size attribute.

MOVS MOVSB MOVSW MOVSD

Move data from string to string
MOVSD 386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks 486	386	286	86	Description
A4	MOVS m8,m8	7	7	5	18	Move byte [(E)SI] to ES:[(E)DI]
A5	MOVS m16,m16	7	7	5	18	Move word [(E)SI] to ES:[(E)DI]
A5	MOVm32,m32	7	7			Move dword [(E)SI] to ES:[(E)DI]
A4	MOVSB	7	7	5	18	Move byte DS:[(E)SI] to ES:[(E)DI]
A5	MOVSW	7	7	5	18	Move word DS:[(E)SI] to ES:[(E)DI]
A5	MOVSD	7	7			Move dword DS:[(E)SI] to ES:[(E)DI]

MOVS copies the byte or word at [(E)SI] to the byte or word at ES:[(E)DI]. The destination operand must be addressable from the ES register; no segment override is possible for the destination. A segment override can be used for the source operand; the default is DS.

The addresses of the source and destination are determined solely by the contents of (E)SI and (E)DI. Load the correct index values into (E)SI and (E)DI before executing the MOVS instruction. MOVSB, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS instructions.

After the data is moved, both (E)SI and (E)DI are advanced automatically. If the direction flag is 0 (CLD was executed), the registers are incremented; if the direction flag is 1 (STD was executed), the registers are decremented. The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.

MOVS can be preceded by the REP prefix for block movement of CX bytes or words. Refer to the REP instruction for details of this operation.

MOVSX

Move with sign-extend
386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks 486	386	Description
0F BE /r	MOVSX r16,r/m8	3/3	3/6	Move byte to word with sign extend
0F BE /r	MOVSX r32,r/m8	3/3	3/6	Move byte to dword
0F BE /r	MOVSX r32,r/m16	3/3	3/6	Move word to dword

MOVSX reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

MOVZX

Move with zero-extend
386 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks		Description
		486	386	
0F B6 /r	MOVZX r16,r/m8	3/3	3/6	Move byte to word with zero extend
0F B6 /r	MOVZX r32,r/m8	3/3	3/6	Move byte to dword
0F B7 /r	MOVZX r32,r/m16	3/3	3/6	Move word to dword

MOVZX reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

MUL

Unsigned multiplication of AL or AX

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F6 /4	MUL r/m8	13/18, 13/18	9-14/ 12-17	13/16	70-77/76-83+EA	Unsigned multiply (AX [(AL 8 r/m byte)
F7 /4	MUL r/m16	13/26, 13/26	9-22/ 12-25	21/24	118-113/124-139+EA	(DX:AX[AX * r/m word)
F7 /4	MUL r/m32	13/42, 13/42	9-38/ 12-41			Unsigned multiply (EDX: EAX[EAX * r/m dword)

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX: AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1 (386 only).

NEG Two's complement negation

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F6 /3	NEG r/m8	1/3	2/6	2/7	3/16+EA	Two's complement negate r/m byte
F7 /3	NEG r/m16	1/3	2/6	2/7	3/16+EA	Two's complement negate r/m word
F7 /3	NEG r/m32	1/3	2/6			Two's complement negate r/m dword

NEG replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.

NOP No operation

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
90	NOP	1	3	3	3	No operation

NOP performs no operation. NOP is a one-byte instruction that takes up space but affects none of the machine context except (E)IP.

NOP is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

NOT One's complement negation

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F6 /2	NOT r/m8	1/3	2/6	2/7	3/16+EA	Reverse each bit of r/m byte
F7 /2	NOT r/m16	1/3	2/6	2/7	3/16+EA	Reverse each bit of r/m word
F7 /2	NOT r/m32	1/3	2/6	2/7		Reverse each bit of r/m dword

NOT inverts the operand; every 1 becomes a 0, and vice versa.

OR

Logical inclusive OR

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
0C ib	OR AL,imm8	1	2	3	4	OR immediate byte to AL
0D iw	OR AX,imm16	1	2	3	4	OR immediate word to AX
0D id	OR EAX,imm32	1	2			OR immediate dword to EAX
80 /1 ib	OR r/m8,imm8	1/3	2/7	3/7	4/17+EA	OR immediate byte to r/m byte
81 /1 iw	OR r/m16,imm16	1/3	2/7	3/7	4/17+EA	OR immediate word to r/m word
81 /1 id	OR r/m32,imm32	1/3	2/7			OR immediate dword to r/m dword
83 /1 ib	OR r/m16,imm8	1/3	2/7			OR sign-extended immediate byte with r/m word
83 /1 ib	OR r/m32,imm8	1/3	2/7			OR sign-extended immediate byte with r/m dword
08 /r	OR r/m8,r8	1/3	2/6	2/7	3/16+EA	OR byte register to r/m byte
09 /r	OR r/m16,r16	1/3	2/6	2/7	3/16+EA	OR word register to r/m word
09 /r	OR r/m32,r32	1/3	2/6			OR dword register to r/m dword
0A /r	OR r8,r/m8	1/2	2/7	2/7	3/9+EA	OR byte register to r/m byte
0B /r	OR r16,r/m16	1/2	2/7	2/7	3/9+EA	OR word register to r/m word
0B /r	OR r32,r/m32	1/2	2/7			OR dword register to r/m word

OR computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

The optimized form of OR is SETFLAG (see Part 3).

OUT

Output to port

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
E6 ib	OUT imm8,AL	16,pm=11*/31**,vm=29	10,pm=4*/24**	3	10	Output byte AL to immediate port number
E7 ib	OUT imm8,AX	16,pm=11*/31**,vm=29	10,pm=4*/24**	3	10	Output word AX to immediate port number
E7 ib	OUT imm8,EAX	16,pm=11*/31**,vm=29	10,pm=4*/25**			Output dword EAX to immediate port number
EE	OUT DX,AL	16,pm=11*/31**,vm=29	11,pm=5*/25**	3	8	Output byte AL to port number in DX
EF	OUT DX,AX	16,pm=11*/31**,vm=29	11,pm=5*/25**	3	8	Output word AX to port number in DX
EF	OUT DX,EAX	16,pm=11*/31**,vm=29	11,pm=5*/25**			Output dword EAX to port number in DX

* If CPL ≤ IOPL
** If CPL > IOPL or if in virtual 8086 mode

OUT transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with DX as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16 bits.

OUTS OUTSB OUTSW OUTSD

Output string to port

OUTS/OUTSB/OUTSW 80186 and greater

OUTSD 386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks 486	Clocks 386	Clocks 286	Description
6E	OUTS DX,r/m8	17,pm=10*/32**,vm=30	14,pm=8*/28**	5	Output byte [(E)SI] to port in DX
6F	OUTS DX,r/m16	17,pm=10*/32**,vm=30	14,pm=8*/28**	5	Output word [(E)SI] to port in DX
6F	OUTS DX,r/m32	17,pm=10*/32**,vm=30	14,pm=8*/28**		Output dword [(E)SI] to port in DX
6E	OUTSB	17,pm=10*/32**,vm=30	14,pm=8*/28**	5	Output byte DS:[(E)SI] to port in DX
6F	OUTSW	17,pm=10*/32**,vm=30	14,pm=8*/28**	5	Output word DS:[(E)SI] to port number in DX
6F	OUTSD	17,pm=10*/32**,vm=30	14,pm=8*/28**		Output dword DS:[(E)SI] to port in DX

OUTS transfers data from the memory byte, word, or doubleword at the source-index register to the output port addressed by the DX register. If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise, the address-size attribute is 32 bits, and ESI is used for the source-index register.

OUTS does not allow specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the OUTS instruction.

The address of the source data is determined by the contents of source-index register. Load the correct index value into SI or ESI before executing the OUTS instruction.

After the transfer, source-index register is advanced automatically. If the direction flag is 0 (CLD was executed), the source-index register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The amount of the increment or decrement is 1 if a byte is output, 2 if a word is output, or 4 if a doubleword is output.

OUTSB, OUTSW, and OUTSD are synonyms for the byte, word, and doubleword OUTS instructions. OUTS can be preceded by the REP prefix for block output of CX bytes or words. Refer to the REP instruction for details on this operation.

POP

Pop a word from the stack

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
8F /0	POP m16	6	5	5	17+EA	Pop top of stack into memory word
8F /0	POP m32	6	5			Pop top of stack into memory dword
58+rw	POP r16	4	4	5	8	Pop top of stack into word register
58+rd	POP r32	4	4			Pop top of stack into dword register
1F	POP DS	3	7,pm=21	5,pm=20	8	Pop top of stack into DS
07	POP ES	3	7,pm=21	5,pm=20	8	Pop top of stack into ES
17	POP SS	3	7,pm=21	5,pm=20	8	Pop top of stack into SS
0F A1	POP FS	3	7,pm=21			Pop top of stack into FS
0F A9	POP GS	3	7,pm=21			Pop top of stack into GS

POP replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the stack, addressed by SS:SP (address-size attribute of 16 bits) or SS:ESP (address-size attribute of 32 bits). The stack pointer SP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

POP CS is not an instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. In protected mode, loading the selector initiates automatic loading of the descriptor information associated with that selector into the hidden part of the segment register; loading also initiates validation of both the selector and the descriptor information.

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a protection exception. An attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection fault. No memory reference occurs. The saved value of the segment register is null.

A POP SS instruction inhibits all interrupts, including NMI, until after execution of the next instruction. This allows sequential execution of POP SS and POP ESP instructions without danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

Note: Turbo Assembler extends the syntax of the POP instruction to facilitate popping multiple items in sequence. The items popped can include any legal POP value, including registers, immediate values, and memory locations. This feature does not actually affect the code generated.

POPA POPAD POPAW

Pop all general registers
POPA 80186 processors and greater
POPAD 386 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
61	POPA	9	24	19	Pop DI, SI, BP, BX, DX, CX, AX
61	POPAD	9	24		Pop EDI, ESI, EBP, EBX, EDX, ECX, EAX
61	POPAW	9	24	19	Pop DI, SI, BP, BX, DX, CX, AX

POPA pops the eight 16- or 32-bit general registers depending on the segment size. However, the SP value is discarded instead of loaded into SP. POPA reverses a previous PUSHA, restoring the general registers to their values before PUSHA was executed. The first register popped is DI.

POPAD pops the eight 32-bit general registers. The ESP value is discarded instead of loaded into ESP. POPAD reverses the previous PUSHAD, restoring the general registers to their values before PUSHAD was executed. The first register popped is EDI.

POPAW pops WORD-sized registers. (Can only be used for VERSION T320 or higher.)

POPF POPFD POPFW

Pop from stack into FLAGS or EFLAGS register
POPFD 386 processors and greater

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
9D	POPF	9,pm=6	5	5	8	Pop top of stack into FLAGS
9D	POPFD	9,pm=6	5			Pop top of stack into EFLAGS
9D	POPFW	9,pm=6	5	5	8	Pop top of stack into FLAGS.

POPF/POPFD pops the word or doubleword on the top of the stack and stores the value in the flags register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in FLAGS. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in EFLAGS.

Note that bits 16 and 17 of EFLAGS, called VM and RF, respectively, are not affected by POPF or POPFD.

The I/O privilege level is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the I/O privilege level. (Real-address mode is equivalent to privilege level 0.) If a POPF instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

POPFW always pops into FLAGS WORD-style. (Can only be used for VERSION T320 or higher.)

PUSH — Push operand onto the stack

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
FF /6	PUSH m16	4	5	5	16+EA	Push memory word
FF /6	PUSH m32	4	5			Push memory dword
50+ /r	PUSH r16	1	2	3	11	Push register word
50+ /r	PUSH r32	1	2			Push register dword
6A	PUSH imm8	1	2 3			Push immediate byte
68	PUSH imm16	1	2	3		Push immediate word
68	PUSH imm32	1	2			Push immediate dword
0E	PUSH CS	3	2	3	10	Push CS
16	PUSH SS	3	2	3	10	Push SS
1E	PUSH DS	3	2	3	10	Push DS
06	PUSH ES	3	2		10	Push ES
0F A0	PUSH FS	3	2			Push FS
0F A8	PUSH GS	3	2			Push GS

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 386 PUSH ESP instruction pushes the value of the ESP as it existed before the instruction. The 80286 PUSH SP instruction also pushes the value of SP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

Note: Turbo Assembler extends the syntax of the PUSH instruction to facilitate pushing multiple items in sequence. The items pushed can include any legal PUSH value, including registers, immediate values, and memory locations. This feature does not actually affect the code generated. In addition, the PUSH instruction allows constant arguments even when generating code for the 8086 processor. Such instructions are replaced in the object code by a 10-byte sequence that simulates the 80186/286/386 PUSH immediate value instruction.

PUSHA PUSHAD PUSHAW

Push all general registers
PUSHA 80186 processors and greater
PUSHAD 386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks			Description
		486	386	286	
60	PUSHA	11	18	17	Push AX,CX,DX,BX,original SP,BP,SI
60	PUSHAD	11	18		Push EAX,ECX,EDX,EBX
60	PUSHAW	11	18	17	Push AX,CX,DX,BX,original SP,BP,SI

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the stack depending on the segment size. PUSHA decrements the stack pointer (SP) by 16 to hold the eight word values. PUSHAD decrements the stack pointer (ESP) by 32 to hold the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 or 32 new stack bytes in reverse order. The last register pushed is DI or EDI.

PUSHAW always pushes WORD-style. (Can only be used for VERSION T320 or higher.)

PUSHF PUSHFD PUSHFW

Push flags register onto the stack
PUSHFD 386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
9C	PUSHF	4,pm=3	4	3	10	Push FLAGS
9C	PUSHFD	4,pm=3	4			Push EFLAGS
9C	PUSHFW	4,pm=3	4	3	10	Push FLAGS

PUSHF decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; PUSHFD decrements the stack pointer by 4, and the 386 EFLAGS register is copied to the new top of stack which is pointed to by SS:ESP.

PUSHFW always pops WORD-sized registers. (Can only be used for VERSION T320 or higher.)

RCL RCR ROL ROR

Rotate

O	D	I	T	S	Z	A	P	C
*								*

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D0 /2	RCL r/m8,1	3/4	9/10	2/7	2/15+EA	Rotate 9 bits (CF,r/m byte) left once
D2 /2	RCL r/m8,CL	8-30/9-31	9/10	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 9 bits (CF,r/m byte) left CL times
C0 /2 ib	RCL r/m8,imm8	8-30/9-31	9/10	5/8		Rotate 9 bits (CF,r/m byte) left imm8 times
D1 /2	RCL r/m16,1	3/4	9/10	2/7	2/15+EA	Rotate 17 bits (CF,r/m word) left once
D3 /2	RCL r/m16,CL	8-30/9-31	9/10	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 17 bits (CF, r/m word) left CL times
C1 /2 ib	RCL r/m16, imm8	8-30/9-31	9/10	5/8		Rotate 17 bits (CF,r/m word)) left imm8 times
D1 /2	RCL r/m32,1	3/4	9/10			Rotate 33 bits (CF,r/m dword) left once
D3 /2	RCL r/m32,CL	8-30/9-31	9/10			Rotate 33 bits (CF,r/m dword) left CL times
C1 /2 ib	RCL r/m32, imm8	8-30/9-31	9/10			Rotate 33 bits (CF,r/m dword) left, imm8 times
D0 /3	RCR r/m8,1	3/4	9/10	2/7	2/15+EA	Rotate 9 bits (CF,r/m byte) right once
D2 /3	RCR r/m8,CL	8-30/9-31	9/10	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 9 bits (CF,r/m byte) right CL times
C0 /3 ib	RCR r/m8,imm8	8-30/9-31	9/10	5/8		Rotate 9 bits (CF,r/m byte) right imm8 times
D1 /3	RCR r/m16,1	3/4	9/10	2/7	2/15+EA	Rotate 17 bits (CF,r/m word) right once
D3 /3	RCR r/m16,CL	8-30/9-31	9/10	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 17 bits (CF,r/m word) right CL times
C1 /3 ib	RCR r/m16, imm8	8-30/9-31	9/10	5/8		Rotate 17 bits (CF,r/m word) right imm8 times
D1 /3	RCR r/m32,1	3/4	9/10			Rotate 33 bits (CF,r/m dword) right once
D3 /3	RCR r/m32,CL	8-30/9-31	9/10			Rotate 33 bits (CF,r/m dword) right CL times
C1 /3 ib	RCR r/m32, imm8	8-30/9-31	9/10			Rotate 33 bits (CF,r/m dword) right imm8 times
D0 /0	ROL r/m8,1	3/4	3/7	2/7	2/15+EA	Rotate 8 bits r/m byte left once
D2 /0	ROL r/m8,CL	3/4	3/7	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 8 bits r/m byte left CL times
C0 /0 ib	ROL r/m8, imm8	2/4	3/7	5/8		Rotate 8 bits r/m byte left imm8 times
D1 /0	ROL r/m16,1	3/4	3/7	2/7	2/15+EA	Rotate 16 bits r/m word left once
D3 /0	ROL r/m16,CL	3/4	3/7	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 16 bits r/m word left CL times
C1 /0 ib	ROL r/m16, imm8	2/4	3/7	5/8		Rotate 16 bit r/m word left imm8 times
D1 /0	ROL r/m32,1	3/4	3/7			Rotate 32 bits r/m dword left once

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D3 /0	ROL r/m32,CL	3/4	3/7			Rotate 32 bits r/m dword left CL times
C1 /0 ib	ROL r/m32, imm8	2/4	3/7			Rotate 32 bits r/m dword left imm8 times
D0 /1	ROR r/m8,1	3/4	3/7	2/7	2/15+EA	Rotate 8 bits r/m byte right once
D2 /1	ROR r/m8,CL	3/4	3/7	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 8 bits r/m byte right CL times
C0 /1 ib	ROR r/m8, imm8	2/4	3/7	5/8		Rotate 8 bits r/m word right imm8 times
D1 /1	ROR r/m16,1	3/4	3/7	2/7	2/15+EA	Rotate 16 bits r/m word right once
D3 /1	ROR r/m16,CL	3/4	3/7	5/8	8+4 per bit/(20+4 per bit)+EA	Rotate 16 bits r/m word right CL times
C1 /1 ib	ROR r/m16, imm8	2/4	3/7	5/8		Rotate 16 bit r/m word right imm8 times
D1 /1	ROR r/m32,1	3/4	3/7			Rotate 32 bits r/m dword right once
D3 /1	ROR r/m32,CL	3/4	3/7			Rotate 32 bits r/m dword right CL times
C1 /1 ib	ROR r/m32, imm8	2/4	3/7			Rotate 32 bits r/m dword right imm8 times

Add 1 clock to the times shown for each rotate made (80286 only).

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all the bits upward, except for the top bit, which is returned to the bottom. The right rotate instructions do the reverse: The bits shift downward until the bottom bit arrives at the top.

For the RCL and RCR instructions, the carry flag is part of the rotated quantity. RCL shifts the carry flag into the bottom bit and shifts the top bit into the carry flag; RCR shifts the carry flag into the top bit and shifts the bottom bit into the carry flag. For the ROL and ROR instructions, the original value of the carry flag is not a part of the result, but the carry flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction execution time, the 80286/386 does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used. The 8086 does not mask rotation counts. The 386 in virtual 8086 mode does mask rotation counts.

The overflow flag is defined only for the single-rotate forms of the instructions (second operand = 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get OF.

RDMSR

Read from Model Specific Register
Pentium processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks	Description
		Pentium	
0F 32	RDMSR	20-24	Read Model Specific Register indicated by ECX into EDX:EAX

The value in ECX specifies one of the 64-bit Model Specific Registers of the Pentium processor. The content of that Model Specific Register is copied into EDX:EAX. EDX is loaded with the high-order 32 bits, and EAX is loaded with the low-order 32 bits.

The following values are used to select model specific registers on the Pentium processor:

Value (in Hex)	Register Name	Description
00h	Machine Check Address	Stores address of cycle causing the exception.
01h	Machine Check Type	Stores cycle type of cycle causing the exception.

Other values used to preform cache, TLB and BTB testing and performance monitoring, are availible under a non-disclosure agreement from Intel.

Protected mode exceptions: #GP(0) if either the current privilege level is not 0 or the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

Real mode exceptions: #GP if the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

Virtual 8086 mode exceptions: #GP(0) if instruction execution is attempted.

Notes: This instruction must be executed at privilege level 0 or in real-address mode; otherwise a protection exception will be generated.

If less than 64 bits are implemented in a model specific register, the value returned to EDX:EAX, in the locations corrisponding to the unimplemented bits, is unpredictable.

RDMSR is used to read the content of Model-Specific Registers that control functions for testability, execution tracing, performance monitoring and machine check errors. Refer to the Pentium Processor Data Book for more information or contact Intel.

The values 3h, 0Fh, and values above 13h are reserved. Do not execute RDMSR with reserved values in ECX.

RDTSC

(Propriatary instruction. Contact Intel for more information.)
Pentium processors and greater

REP REPE REPZ REPNE REPNZ

Repeat following string operation

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F3 6C	REP INS r/m8,DX	16+8(E)CX, pm=10+8(E)CX*1/ 30+8(E)CX*2,VM= 29+8(E)CX	13+6*(E)CX, pm=7+6*(E)CX/ 27+6*1*(E)CX*2	5+4*CX		Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS r/m16,DX	16+8(E)CX, pm=10+8(E)CX*1/ 30+8(E)CX*2,VM= 29+8(E)CX	13+6*(E)CX, pm=7+6*(E)CX/ 27+6*1*(E)CX*2	5+4*CX		Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS r/m32,DX	16+8(E)CX, pm=10+8(E)CX*1/ 30+8(E)CX*2,VM= 29+8(E)CX	13+6*(E)CX, pm=7+6*(E)CX/ 27+6*1*(E)CX*2			Input (E)CX dwords from port DX into ES:[(E)DI]
F3 A4	REP MOVS m8,m8	5*3,13*4,12+3(E) CX*5	5+4*(E)CX	5+4*CX	9+17*CX	Move (E)CX bytes from [(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS m16,m16	5*3,13*4,12+3(E) CX*5	5+4*(E)CX	5+4*CX	9+17*CX	Move (E)CX words from [(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS m32,m32	5*3,13*4,12+3(E) CX*5	5+4*(E)CX			Move (E)CX dwords from [(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX,r/m8	17+5(E)CX, pm=11+5(E)CX*1/ 31+5(E)CX*2	5+12*(E)CX, pm=6+5*(E) CX/26+5*1*(E) CX*2	5+4*CX		Output (E)CX bytes from [(E)SI] to port DX
F3 6F	REP OUTS DX,r/m16	17+5(E)CX, pm=11+5(E)CX*1/ 31+5(E)CX*2	5+12*(E)CX, pm=6+5*(E) CX/26+5*1*(E) CX*2	5+4*CX		Output (E)CX words from [(E)SI] to port DX
F3 6F	REP OUTS DX,r/m32	17+5(E)CX, pm=11+5(E)CX*1/ 31+5(E)CX*2	5+12*(E)CX, pm=6+5*(E) CX/26+5*1*(E) CX*2			Output(E)CX dwords from [(E)SI] to port DX
F2 AC	REP LODS m8	5*3,7+4(E)CX*6				Load (E)CX bytes from [(E)SI] to AL
F2 AD	REP LODS m16	5*3,7+4(E)CX*6				Load (E)CX words from [(E)SI] to AX
F2 AD	REP LODS m32	5*3,7+4(E)CX*6				Load (E)CX dwords from [(E)SI] to EAX
F3 AA	REP STOS m8	5*3,7+4(E)CX*6	5+5*(E)CX	4+3*CX	9+10*CX	Fill (E)CX bytes at ES:[(E)DI] with AL

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F3 AB	REP STOS m16	5^{*3},7+4(E)CX^{*6}	5+5*(E)CX	4+3*CX	9+10*CX	Fill (E)CX words at ES:[(E)DI] with AX
F3 AB	REP STOS m32	5^{*3},7+4(E)CX^{*6}	5+5*(E)CX			Fill (E)CX dwords at ES:[(E)DI] with EAX
F3 A6	REPE CMPS m8,m8	5^{*3},7+7(E)CX^{*6}	5+9*N	5+9*N	9+22*N	Find nonmatching bytes in ES:[(E)DI] and [(E)SI]
F3 A7	REPE CMPS m16,m16	5^{*3},7+7(E)CX^{*6}	5+9*N	5+9*N	9+22*N	Find nonmatching words in ES:[(E)DI] and [(E)SI]
F3 A7	REPE CMPS m32,m32	5^{*3},7+7(E)CX^{*6}	5+9*N			Find nonmatching dwords in ES:[(E)DI] and [(E)SI]
F3 AE	REPE SCAS m8	5^{*3},7+5(E)CX^{*6}	5+8*N	5+8*N	9+15*N	Find non-AL byte starting at ES:[(E)DI]
F3 AF	EPE SCAS m16	5^{*3},7+5(E)CX^{*6}	5+8*N	5+8*N	9+15*N	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS m32	5^{*3},7+5(E)CX^{*6}	5+8*N			Find non-EAX dword starting at ES:[(E)DI]
F2 A6	REPNE CMPS m8,m8	5^{*3},7+7(E)CX^{*6}	5+9*N	5+9*N	9+22*N	Find matching bytes in ES:[(E)DI] and [(E)SI]
F2 A7	REPNE CMPS m16,m16	5^{*3},7+7(E)CX^{*6}	5+9*N	5+9*N	9+22*N	Find matching words in ES:[(E)DI] and [(E)SI]
F2 A7	REPNE CMPS m32,m32	5^{*3},7+7(E)CX^{*6}	5+9*N			Find matching dwords in ES:[(E)DI] and [(E)SI]
F2 AE	REPNE SCAS m8	5^{*3},7+5(E)CX^{*6}	5+8*N	5+8*N	9+15*N	Find AL
F2 AF	REPNE SCAS m16	5^{*3},7+5(E)CX^{*6}	5+8*N	5+8*N	9+15*N	Find AX
F2 AF	REPNE SCAS m32	5^{*3},7+5(E)CX^{*6}	5+8*N			Find EAX

*1 If CPL ≤ IOPL
*2 If CPL > IOPL
*3 If (E) CX = 0
*4 If (E) CX = 1
*5 If (E) CX 1
*6 If (E) CX 0

REP, REPE (repeat while equal), and REPNE (repeat while not equal) are prefixes that are applied to string operations. Each prefix causes the string instruction that follows to be repeated the number of times indicated in the count register or (for REPE and REPNE) until the indicated condition in the zero flag is no longer met.

Synonymous forms of REPE and REPNE are REPZ and REPNZ, respectively.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

The precise action for each iteration is as follows:

1. If the address-size attribute is 16 bits, use CX for the count register; if the address-size attribute is 32 bits, use ECX for the count register.

2. Check CX. If it is zero, exit the iteration, and move to the next instruction.

3. Acknowledge any pending interrupts.

4. Perform the string operation once.

5. Decrement CX or ECX by one; no flags are modified.

6. Check the zero flag if the string operation is SCAS or CMPS. If the repeat condition does not hold, exit the iteration and move to the next instruction. Exit the iteration if the prefix is REPE and ZF is 0 (the last comparison was not equal), or if the prefix is REPNE and ZF is one (the last comparison was equal).

7. Return to step 1 for the next iteration.

Repeated CMPS and SCAS instructions can be exited if the count is exhausted or if the zero flag fails the repeat condition. These two cases can be distinguished by using either the JCXZ instruction, or by using the conditional jumps that test the zero flag (JZ, JNZ, and JNE).

RET Return from procedure

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
C3	RET	5	10+m	11	16	Return (near) to caller
CB	RET	13,pm=18	18+m,pm= 32+m	15,pm=25	26	Return (far) to caller, same privilege
CB	RET	13,pm=33	pm=68	55		Return (far)
C2 iw	RET imm16	5	10+m	11	20	Return (near)
CA iw	RET imm16	14,pm=17	18+m,pm= 32+m	15,pm=25	25	Return (far) pop imm16 bytes
CA iw	RET imm16	14,pm=33	pm=68	55		Return (far)

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL.

The optional numeric parameter to RET gives the number of stack bytes (OperandMode = 16) or words (OperandMode = 32) to be released after the return address is popped. These items are typically used as input parameters to the procedure called.

For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged. For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

In real mode, CS and IP are loaded directly. In protected mode, an intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

The DS, ES, FS, and GS segment registers can be set to 0 by the RET instruction during an interlevel transfer. If these registers refer to segments that cannot be used by the new privilege level, they are set to 0 to prevent unauthorized access from the new privilege level.

RSM

Resume from System Management Mode
Pentium processors and greater

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

Opcode	Instruction	Clocks	Description
		Pentium	
0F AA	RSM	83	Resume operation of interrupted program.

Resume operation of a program by a System Management Mode (SMM) interrupt. The processor state is restored from the dump created upon entrance to SMM. Note, however, that the contents of the model-specific registers are not affected. The processor leaves SMM and returns control to the interrupted application or operating system. If the processor detects any invalid state information, it enters the shutdown state. This happens in any of the following situations:

- The value stored in the State Dump Base field is not a 32 Kbyte aligned address.
- Any reserved bit in CR4 is set to 1.
- Any combination of bits in CR0 is illegal; namely, (PG=1 and PE=0) or (NW=1 and CD=0).

Protected mode, Real mode, and Virtual 8086 mode exception: #UD if an attempt is made to execute this instruction when the processor is not in SMM.

Notes: for more information about SMM and the behavior of the RSM instruction, see the Pentium Processor User's Manual (availible from Intel)

SAHF

Store AH into Flags

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
9E	SAHF	2	3	2	4	Store AH flags SF ZF xx AF xx PF xx CF

SAHF loads the flags listed above with values from the AH register, from bits 7, 6, 4, 2 and 0, respectively.

SAL
SAR
SHL
SHR

Shift instructions

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D0 /4	SAL r/m8,1	3/4	3/7	2/7	2/15+EA	Multiply r/m byte by 2
D2 /4	SAL r/m8,CL	3/4	3/7	5/8	8+4 per bit/(20+4 per bit)+EA	Multiply r/m byte by 2, CL times
C0 /4 ib	SAL r/m8,imm8	2/4	3/7	5/8		Multiply r/m byte by 2
D1 /4	SAL r/m16,1	3/4	3/7	2/7	2/15+EA	Multiply r/m word by 2
D3 /4	SAL r/m16,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Multiply r/m word by 2, CL times
C1 /4 ib	SAL r/m16,imm8	2/4	3/7	5/8		Multiply r/m word by 2
D1 /4	SAL r/m32,1	3/4	3/7			Multiply r/m dword by 2
D3 /4	SAL r/m32,CL	3/4	3/7			Multiply r/m dword by 2
C1 /4 ib	SAL r/m32,imm8	2/4	3/7			Multiply r/m dword by 2
D0 /7	SAR r/m8,1	3/4	3/7	2/7	2/15+EA	Signed divide** r/m byte by 2
D2 /7	SAR r/m8,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Signed divide** r/m byte by 2
C0 /7 ib	SAR r/m8,imm8	2/4	3/7	5/8		Signed divide** r/m byte by 2
D1 /7	SAR r/m16,1	3/4	3/7	2/7	2/15+EA	Signed divide** r/m word by 2
D3 /7	SAR r/m16,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Signed divide** r/m word by 2
C1 /7 ib	SAR r/m16,imm8	2/4	3/7	5/8		Signed divide** r/m word by 2
D1 /7	SAR r/m32,1	3/4	3/7			Signed divide** r/m dword by 2
D3 /7	SAR r/m32,CL	3/4	3/7			Signed divide** r/m dword by 2, CL times
C1 /7	SAR r/m32,imm8	2/4	3/7			Signed divide** r/m dword by 2
D0 /4	SHL r/m8,1	3/4	3/7	2/7	2/15+EA	Multiply r/m byte by 2
D2 /4	SHL r/m8,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Multiply r/m byte by 2, CL times
C0 /4 ib	SHL r/m8,imm8	2/4	3/7	5/8		Multiply r/m byte by 2
D1 /4	SHL r/m16,1	3/4	3/7	2/7	2/15+EA	Multiply r/m word by 2
D3 /4	SHL r/m16,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Multiply r/m word by 2, CL times
C1 /4 ib	SHL r/m16,imm8	2/4	3/7	5/8		Multiply r/m word by 2
D1 /4	SHL r/m32,1	3/4	3/7			Multiply r/m dword by 2
D3 /4	SHL r/m32,CL	3/4	3/7			Multiply r/m dword by 2
C1 /4 ib	SHL r/m32,imm8	2/4	3/7			Multiply r/m dword by 2

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D0 /5	SHR r/m8,1	3/4	3/7	2/7	2/15+EA	Unsigned divide r/m byte by 2
D2 /5	SHR r/m8,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Unsigned divide r/m byte by 2
C0 /5 ib	SHR r/m8,imm8	2/4	3/7	5/8		Unsigned divide r/m byte by 2
D1 /5	SHR r/m16,1	3/4	3/7	2/7	2/15+EA	Unsigned divide r/m word by 2
D3 /5	SHR r/m16,CL	3/4	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Unsigned divide r/m word by 2
C1 /5 ib	SHR r/m16,imm8	2/4	3/7	5/8		Unsigned divide r/m word by 2
D1 /5	SHR r/m32,1	3/4	3/7			Unsigned divide r/m dword by 2
D3 /5	SHR r/m32,CL	3/4	3/7			Unsigned divide r/m dword by 2
C1 /5 ib	SHR r/m32,imm8	2/4	3/7			Unsigned divide r/m dword by 2

**Not the same division as IDIV; rounding is toward negative infinity.

SAL (or its synonym, SHL) shifts the bits of the operand upward. The high-order bit is shifted into the carry flag, and the low-order bit is set to 0.

SAR and SHR shift the bits of the operand downward. The low-order bit is shifted into the carry flag. The effect is to divide the operand by 2. SAR performs a signed divide with rounding toward negative infinity (not the same as IDIV); the high-order bit remains the same. SHR performs an unsigned divide; the high-order bit is set to 0.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum execution time, the 80286/386 does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, OF is set to 0 if the high bit of the answer is the same as the result of the carry flag (that is, the top two bits of the original operand were the same); OF is set to 1 if they are different. For SAR, OF is set to 0 for all single shifts. For SHR, OF is set to the high-order bit of the original operand.

SBB Integer subtraction with borrow

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
1C ib	SBB AL,imm8	1	2	3	4	Subtract with borrow immediate byte from AL
1D iw	SBB AX,imm16	1	2	3	4	Subtract with borrow immediate word from AX

Opcode	Instruction	Clocks				Description
		486	386	286	86	
1D id	SBB EAX,imm32	1	2			Subtract with borrow immediate dword from EAX
80 /3 ib	SBB r/m8,imm8	1/3	2/7	3/7	4/17+EA	Subtract with borrow immediate byte from r/m byte
81 /3 iw	SBB r/m16,imm16	1/3	2/7	3/7	4/17+EA	Subtract with borrow immediate word from r/m word
81 /3 id	SBB r/m32,imm32	1/3	2/7			Subtract with borrow immediate dword from r/m dword
83 /3 ib	SBB r/m16,imm8	1/3	2/7	3/7	4/17+EA	Subtract with borrow sign-extended immediate byte from r/m word
83 /3 ib	SBB r/m32,imm8	1/3	2/7			Subtract with borrow sign-extended immediate byte from r/m dword
18 /r	SBB r/m8,r8	1/3	2/6	2/7	3/16+EA	Subtract with borrow byte register from r/m byte
19 /r	SBB r/m16,r16	1/3	2/6	2/7	3/16+EA	Subtract with borrow word register from r/m word
19 /r	SBB r/m32,r32	1/3	2/6			Subtract with borrow dword register from r/m dword
1A /r	SBB r8,r/m8	1/2	2/7	2/7	3/9+EA	Subtract with borrow byte register from r/m byte
1B /r	SBB r16,r/m16	1/2	2/7	2/7	3/9+EA	Subtract with borrow word register from r/m word
1B /r	SBB r32,r/m32	1/2	2/7			Subtract with borrow dword register from r/m dword

SBB adds the second operand (DEST) to the carry flag (CF) and subtracts the result from the first operand (SRC). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

SCAS SCASB SCASW SCASD

Compare string data

SCASD 386 processors and greater

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
AE	SCAS m8	6	7	7	15	Compare bytes AL - ES:[DI]
AF	SCAS m16	6	7	7	15	Compare words AX - ES: [DI]
AF	SCAS m32	6	7			Compare dwords EAX - ES: [DI]
AE	SCASB	6	7	7	15	Compare bytes AL - ES:[DI]
AF	SCASW	6	7	7	15	Compare words AX - ES: [DI]
AF	SCASD	6	7			Compare dwords EAX - ES: [DI]

SCAS subtracts the memory byte or word at the destination register from the AL, AX or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible.

If the address-size attribute for this instruction is 16 bits, DI is used as the destination register; otherwise, the address-size attribute is 32 bits and EDI is used.

The address of the memory data being compared is determined solely by the contents of the destination register, not by the operand to SCAS. The operand validates ES segment addressability and determines the data type. Load the correct index value into DI or EDI before executing SCAS.

After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (CLD was executed), the destination register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The increments or decrements are by 1 if bytes are compared, by 2 if words are compared, or by 4 if doublewords are compared.

SCASB, SCASW, and SCASD are synonyms for the byte, word and doubleword SCAS instructions that don't require operands. They are simpler to code, but provide no type or segment checking.

SCAS can be preceded by the REPE or REPNE prefix for a block search of CX or ECX bytes or words. Refer to the REP instruction for further details.

SETcc

Byte set on condition
386 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks 486	Clocks 386	Description
0F 97	SETA r/m8	4/3	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE r/m8	4/3	4/5	Set byte if above or equal (CF=0)
0F 92	SETB r/m8	4/3	4/5	Set byte if below (CF=1)
0F 96	SETBE r/m8	4/3	4/5	Set byte if below or equal (CF=1 or ZF=1)
0F 92	SETC r/m8	4/3	4/5	Set if carry (CF=1)
0F 94	SETE r/m8	4/3	4/5	Set byte if equal (ZF=1)
0F 9F	SETG r/m8	4/3	4/5	Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE r/m8	4/3	4/5	Set byte if greater or equal (SF=OF)
0F 9C	SETL r/m8	4/3	4/5	Set byte if less (SF<>OF)
0F 9E	SETLE r/m8	4/3	4/5	Set byte if less or equal (ZF=1 and SF<>OF)
0F 96	SETNA r/m8	4/3	4/5	Set byte if not above (CF=1)
0F 92	SETNAE r/m8	4/3	4/5	Set byte if not above or equal (CF=1)
0F 93	SETNB r/m8	4/3	4/5	Set byte if not below (CF=0)
0F 97	SETNBE r/m8	4/3	4/5	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC r/m8	4/3	4/5	Set byte if not carry (CF=0)
0F 95	SETNE r/m8	4/3	4/5	Set byte if not equal (ZF=0)
0F 9E	SETNG r/m8	4/3	4/5	Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE r/m8	4/3	4/5	Set byte if not greater or equal (SF<>OF)
0F 9D	SETNL r/m8	4/3	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE r/m8	4/3	4/5	Set byte if not less or equal (ZF=1 and SF<>OF)
0F 91	SETNO r/m8	4/3	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP r/m8	4/3	4/5	Set byte if not parity (PF=0)

Opcode	Instruction	Clocks		Description
		486	386	
0F 99	SETNS r/m8	4/3	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ r/m8	4/3	4/5	Set byte if not zero (ZF=0)
0F 90	SETO r/m8	4/3	4/5	Set byte if overflow (OF=1)
0F 9A	SETP r/m8	4/3	4/5	Set byte if parity (PF=1)
0F 9A	SETPE r/m8	4/3	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO r/m8	4/3	4/5	Set byte if parity odd (PF=0)
0F 98	SETS r/m8	4/3	4/5	Set byte if sign (SF=1)
0F 94	SETZ r/m8	4/3	4/5	Set byte if zero (ZF=1)

SETcc stores a byte containing 1 at the destination specified by the effective address or register if the condition is met, or a 0 byte if the condition is not met.

SGDT SIDT

Store global/interrupt descriptor table
80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 01 /0	SGDT m	10	9	11	Store GDTR to m
0F 01 /1	SIDT m	10	9	12	Store IDTR to m

SGDT/SIDT copies the contents of the descriptor table register to the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 16 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined. Otherwise, if the operand-size attribute is 32 bits, the next four bytes are assigned the 32-bit BASE field of the register.

SGDT and SIDT are used only in operating system software; they are not used in application programs.

SHLD

Double precision shift left
386 processors and greater

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

Opcode	Instruction	Clocks		Description
		486	386	
0F A4	SHLD r/m16,r16,imm8	2/3	3/7	r/m16 gets SHL of r/m16 concatenated with r16
0F A4	SHLD r/m32,r32,imm8	2/3	3/7	r/m32 gets SHL of r/m32 concatenated with r32
0F A5	SHLD r/m16,r16,CL	2/3	3/7	r/m16 gets SHL of r/m16 concatenated with r16
0F A5	SHLD r/m32,r32,CL	2/3	3/7	r/m32 gets SHL of r/m32 concatenated with r32

SHLD shifts the first operand provided by the r/m field to the left as many bits as specified by the count operand. The second operand (r16 or r32) provides the bits to shift in from the right (starting with bit 0). The result is stored back into the r/m operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for multiprecision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CF is set to the value of the last bit shifted out. OF and AF are left undefined.

SHRD

Double precision shift right
386 processors and greater

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

Opcode	Instruction	Clocks		Description
		486	386	
OF AC	SHRD r/m16,r16,imm8	2/3	3/7	r/m16 gets SHR of r/m16 concatenated with r16
OF AC	SHRD r/m32,r32,imm8	2/3	3/7	r/m32 gets SHR of r/m32 concatenated with r32
OF AD	SHRD r/m16,r16,CL	3/4	3/7	r/m16 gets SHR of r/m16 concatenated with r16
OF AD	SHRD r/m32,r32,CL	3/4	3/7	r/m32 gets SHR of r/m32 concatenated with r32

SHRD shifts the first operand provided by the r/m field to the right as many bits as specified by the count operand. The second operand (r16 or r32) provides the bits to shift in from the left (starting with bit 31). The result is stored back into the r/m operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified register, the operation is useful for multi-precision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CF is set to the value of the last bit shifted out. OF and AF are left undefined.

SLDT

Store local descriptor table register
80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 00 /0	SLDT r/m16	2/3	pm=2/2	2/3	Store LDTR to EA word

SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the global descriptor table.

SLDT is used only in operating system software. It is not used in application programs.

SMSW

Store machine status word
80286 and greater protected mode only

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

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 01 /4	SMSW r/m16	2/3	2/3,pm=2/2	2/3	Store machine status word to EA word

SMSW stores the machine status word (part of CR0) in the two-byte register or memory location indicated by the effective address operand.

STC

Set carry flag

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
F9	STC	2	2	2	2	Set carry flag

STC sets the carry flag to 1.

STD

Set direction flag

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
FD	STD	2	2	2	2	Set direction flag so (E)SI or (E)DI decrement

STD sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI, on which they operate.

STI — Set interrupt enable flag

O	D	I	T	S	Z	A	P	C
		1						

Opcode	Instruction	Clocks				Description
		486	386	286	86	
FB	STI	5	3	2	2	Set interrupt flag

STI sets the interrupt flag to 1. The CPU then responds to external interrupts after executing the next instruction if the next instruction allows the interrupt flag to remain enabled. If external interrupts are disabled and you code STI, RET (such as at the end of a subroutine), the RET is allowed to execute before external interrupts are recognized. Also, if external interrupts are disabled and you code STI, CLI, then external interrupts are not recognized because the CLI instruction clears the interrupt flag during its execution.

STOS STOSB STOSW STOSD — Store string data

STOSD 386 processors and greater

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
AA	STOS m8	5	4	3	11	Store AL in byte ES:[(E)DI]
AB	STOS m16	5	4	3	11	Store AX in word ES:[(E)DI]
AB	STOS m32	5	4			Store EAX in dword ES:[(E)DI]
AA	STOSB	5	4	3	11	Store AL in byte ES:[(E)DI]
AB	STOSW	5	4	3	11	Store AX in word ES:[(E)DI]
AB	STOSD	5	4			Store EAX in dword ES:[(E)DI]

STOS transfers the contents of the AL, AX, or EAX register to the memory byte, word, or doubleword given by the destination register relative to the ES segment. The destination register is DI for an address-size attribute of 16 bits or EDI for an address-size attribute of 32 bits.

The destination operand must be addressable from the ES register. A segment override is not possible.

The address of the destination is determined by the contents of the destination register, not by the explicit operand of STOS. This operand is used only to validate ES segment addressability and to determine the data type. Load the correct index value into the destination register before executing STOS.

After the transfer is made, DI is automatically updated. If the direction flag is 0 (CLD was executed), DI is incremented; if the direction flag is 1 (STD was executed), DI is decremented. DI is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored, or by 4 if a doubleword is stored.

STOSB, STOSW, and STOSD are synonyms for the byte, word, and double-word STOS instructions, that do not require an operand. They are simpler to use, but provide no type or segment checking.

STOS can be preceded by the REP prefix for a block fill of CX or ECX bytes, words, or doublewords. Refer to the REP instruction for further details.

STR

Store task register

80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 00 /1	STR r/m16	2/3	pm=23/27	2/3	Load EA word into task register

The contents of the task register are copied to the two-byte register or memory location indicated by the effective address operand.

STR is used only in operating system software. It is not used in application programs.

SUB

Integer Subtraction

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
2C ib	SUB AL,imm8	1	2	3	4	Subtract immediate byte from AL
2D iw	SUB AX,imm16	1	2	3	4	Subtract immediate word from AX
2D id	SUB EAX,imm32	1	2			Subtract immediate dword from EAX
80 /5 ib	SUB r/m8,imm8	1/3	2/7	3/7	4/17+EA	Subtract immediate byte from r/m byte
81 /5 iw	SUB r/m16,imm16	1/3	2/7	3/7	4/17+EA	Subtract immediate word from r/m word
81 /5 id	SUB r/m32,imm32	1/3	2/7			Subtract immediate dword from r/m dword
83 /5 ib	SUB r/m16,imm8	1/3	2/7	3/7	4/17+EA	Subtract sign-extended immediate byte from r/m word
83 /5 ib	SUB r/m32,imm8	1/3	2/7			Subtract sign-extended immediate byte from r/m dword
28 /r	SUB r/m8,r8	1/3	2/6	2/7	3/16+EA	Subtract byte register from r/m byte
29 /r	SUB r/m16,r16	1/3	2/6	2/7	3/16+EA	Subtract word register from r/m word
29 /r	SUB r/m32,r32	1/3	2/6			Subtract dword register from r/m dword
2A /r	SUB r8,r/m8	1/2	2/7	2/7	3/9+EA	Subtract EA byte from byte register
2B /r	SUB r16,r/m32	1/2	2/7	2/7	3/9+EA	Subtract EA word from word register
2B /r	SUB r32,r/m32	1/2	2/7			Subtract EA dword from dword register

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

TEST — Logical compare

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
A8 ib	TEST AL,imm8	1	2	3	4	And immediate byte with AL
A9 iw	TEST AX,imm16	1	2	3	4	And immediate word with AX
A9 id	TEST EAX,imm32	1	2			And immediate dword with EAX
F6 /0 ib	TEST r/m8,imm8	1/2	2/5	3/6	5/11+EA	And immediate byte with r/m byte
F7 /0 iw	TEST r/m16,imm16	1/2	2/5	3/6	5/11+EA	And immediate word with r/m word
F7 /0 id	TEST r/m32,imm32	1/2	2/5			And immediate dword with r/m dword
84 /r	TEST r/m8,r8	1/2	2/5	2/6	3/9+EA	And byte register with r/m byte
85 /r	TEST r/m16,r16	1/2	2/5	2/6	3/9+EA	And word register with r/m word
85 /r	TEST r/m32,r32	1/2	2/5			And dword register with r/m dword

TEST computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

The optimized form of TEST is TESTFLAG (see Part 3).

VERR VERW — Verify a segment for reading or writing 80286 and greater protected mode only

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks			Description
		486	386	286	
0F 00 /4	VERR r/m16	11/11	pm=10/11	14/16	Set ZF=1 if segment can be read
0F 00 /5	VERW r/m16	11/11	pm=15/16	14/16	Set ZF=1 if segment can be written

The two-byte register or memory operand of VERR and VERW contains the value of a selector. VERR and VERW determine whether the segment denoted by the selector is reachable from the current privilege level and whether the segment is readable (VERR) or writable (VERW). If the segment is accessible, the zero flag is set to 1; if the segment is not accessible, the zero flag is set to 0. To set ZF, the following conditions must be met:

- The selector must denote a descriptor within the bounds of the table (GDT or LDT); the selector must be "defined."
- The selector must denote the descriptor of a code or data segment (not that of a task state segment, LDT, or a gate).
- For VERR, the segment must be readable. For VERW, the segment must be a writable data segment.
- If the code segment is readable and conforming, the descriptor privilege level (DPL) can be any value for VERR. Otherwise, the DPL must be greater than or equal to (have less or the same privilege as) both the current privilege level and the selector's RPL.

The validation performed is the same as if the segment were loaded into DS, ES, FS, or GS, and the indicated access (read or write) were performed. The zero flag receives the result of the validation. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

WAIT

Wait until BUSY# pin is inactive (HIGH)

O D I T S Z A P C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
9B	WAIT	1-3	6	3	4+5n	Wait until BUSY pin is inactive (HIGH)

WAIT suspends execution of CPU instructions until the BUSY# pin is inactive (high). The BUSY# pin is driven by the 80x87 numeric processor extension.

WBINVD

Write-back and invalidate cache
i486 processors and greater

O D I T S Z A P C

Opcode	Instruction	Clock	Description
		486	
0F 09	WBINVD	5	Write-back and invalidate entire cache

The internal cache is flushed, and a special-function bus cycle is issued which indicates that the external cache should write-back its contents to main memory. Another special-function bus cycle follows, directing the external cache to flush itself.

Note: This instruction is implementation-dependent; its function might be implemented differently on future Intel processors. It is the responsibility of the hardware to respond to the external cache write-back and flush indications.

WRMSR Write to Model Specific Register
Pentium processors and greater

O D I T S Z A P C

Opcode	Instruction	Clocks	Description
		Pentium	
0F 30	WRMSR	30-45	Write the value in EDX:EAX to Model Specific Register indicated by ECX.

The value in ECX specifies one of the 64-bit Model Specific Registers of the Pentium processor. The contents of EDX:EAX is copied into that Model Specific Register. The high-order 32 bits are copied from EDX and the low-order 32 bits arecopied from EAX.

The following values are used to select model specific registers on the Pentium processor:

Value (in Hex)	Register Name	Description
00h	Machine Check Address	Stores address of cycle causing the exception.
01h	Machine Check Type	Stores cycle type of cycle causing the exception.

Other values used to preform cache, TLB and BTB testing and performance monitoring, are availible under a non-disclosure agreement from Intel.

Protected mode exceptions: #GP(0) if either the current privilege level is not 0 or the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

Real mode exceptions: #GP if the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

Virtual 8086 mode exceptions: #GP(0) if instruction execution is attempted.

Notes: This instruction must be executed at privilege level 0 or in real-address mode; otherwise a protection exception will be generated.

Always set undefined or reserved bits to the value previously read.

WRMSR is used to write the content of Model-Specific Registers that control functions for testability, execution tracing, performance monitoring and machine check errors. Refer to the Pentium Processor Data Book for more information or contact Intel.

The values 3h, 0Fh, and values above 13h are reserved. Do not execute WRMSR with reserved values in ECX.

XADD Exchange and add

i486 processors and greater

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

Opcode	Instruction	Clock	Description
		486	
0F C0/r	XADD r/m8,r8	3/4	Exchange byte register and r/m byte; load sum into r/m byte.
0F C1/r	XADD r/m16,r168	3/4	Exchange word register and r/m word; load sum into r/m word.
0F C1/r	XADD r/m32,r32	3/4	Exchange dword register and r/m dword; load sum into r/m dword.

The XADD instruction loads DEST into SRC, and then loads the sum of DEST and the original value of SRC into DEST.

DEST is the destination operand; SRC is the source operand.

Protected mode exceptions: #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for an unaligned memory reference if the current privilege level is 3.

Real address mode exceptions: interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFh.

Virtual 8086 mode exceptions: same exception as in real-address mode; same #PF and #AC exceptions as in protected mode.

XCHG Exchange memory/register with register

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
86 /r	XCHG r/m8,r8	3/5	3/5	3/5	4/17+EA	Exchange byte register with EA byte
86 /r	XCHG r8,r/m8	3/5	3/5	3/5	4/17+EA	Exchange byte with EA byte register
87 /r	XCHG r/m16,r16	3/5	3/5	3/5	4/17+EA	Exchange word register with EA word
87 /r	XCHG r16,r/m16	3/5	3/5	3/5	4/17+EA	Exchange word register with EA word
87 /r	XCHG r/m32,r32	3/5	3/5			Exchange dword register with EA dword
87 /r	XCHG r32,r/m32	3/5	3/5			Exchange dword register with EA dword
90+ r	XCHG AX,r16	3	3	3	3	Exchange word register with AX

Opcode	Instruction	Clocks				Description
		486	386	286	86	
90+ r	XCHG r16,AX	3	3	3	3	Exchange word register with AX
90+ r	XCHG EAX,r32	3	3			Exchange dword register with EAX
90+ r	XCHG r32,EAX	3	3			Exchange dword register with EAX

XCHG exchanges two operands. The operands can be in either order. If a memory operand is involved, BUS LOCK is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

XLAT XLATB

Table look-up translation

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks				Description
		486	386	286	86	
D7	XLAT m8	4	5	5	11	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	4	5	5	11	Set AL to memory byte DS:[(E)BX + unsigned AL]

XLAT changes the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX (for an address-size attribute of 16 bits) or DS:EBX (for an address-size attribute of 32 bits).

The operand to XLAT allows for the possibility of a segment override. XLAT uses the contents of BX even if they differ from the offset of the operand. The offset of the operand should have been moved into BX/EBX with a previous instruction.

The no-operand form, XLATB, can be used if the BX/EBX table will always reside in the DS segment.

XOR

Logical exclusive OR

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

Opcode	Instruction	Clocks				Description
		486	386	286	86	
34 ib	XOR AL,imm8	1	2	3	4	Exclusive-OR immediate byte to AL
35 iw	XOR AX,imm16	1	2	3	4	Exclusive-OR immediate word to AX
35 id	XOR EAX,imm32	1	2			Exclusive-OR immediate dword to EAX
80 /6 ib	XOR r/m8,imm8	1/3	2/7	3/7	4/17+EA	Exclusive-OR immediate byte to r/m byte
81 /6 iw	XOR r/m16,imm16	1/3	2/7	3/7	4/17+EA	Exclusive-OR immediate word to r/m word
81 /6 id	XOR r/m32,imm32	1/3	2/7			Exclusive-OR immediate dword to r/m dword
83 /6 ib	XOR r/m16,imm8	1/3	2/7			XOR sign-extended immediate byte to r/m word

Opcode	Instruction	Clocks				Description
		486	386	286	86	
83 /6 ib	XOR r/m32,imm8	1/3	2/7			XOR sign-extended immediate byte to r/m dword
30 /r	XOR r/m,r8	1/3	2/6	2/7	3/16+EA	Exclusive-OR byte register to r/m byte
31 /r	XOR r/m16,r16	1/3	2/6	2/7	3/16+EA	Exclusive-OR word register into r/m word
31 /r	XOR r/m32,r32	1/3	2/6			Exclusive-OR dword register to r/m dword
32 /r	XOR r8,r/m8	1/2	2/7	2/7	3/9+EA	Exclusive-OR r/m byte to byte register
33 /r	XOR r16,r/m16	1/2	2/7	2/7	3/9+EA	Exclusive-OR r/m word to word register
33 /r	XOR r32,r/m32	1/2	2/7			Exclusive-OR to r/m dword to dword register

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

The optimized form of XOR is FLIPFLAG (see Part 3).

P A R T 5

Coprocessor instructions

This part lists the 80x87 instructions in alphabetical order.

There is one entry for each combination of operand types that can be coded with the mnemonic. The following table explains the operand identifiers used in this section:

Identifier	Explanation
ST	Stack top; the register currently at the top of the stack.
ST(1)	A register in the stack *i(0≤i≤7)* stack elements from the top. ST(1) is the next-on-stack register, ST(2) is below ST(1), etc.
m32real	A short real (32 bits) number in memory.
m64real	A long real (64 bits) number in memory.
m80real	A temporary real (80 bits) number in memory.
m80dec	A packed decimal integer (18 digits, 10 bytes) in memory.
m16int	A word binary integer (16 bits) in memory.
m32int	A short binary integer (32 bits) in memory.
m64int	A long binary integer (64 bits) in memory.
m*xx*byte	A memory area *xx* bytes long.

Here is a summary of the possible exceptions each instruction can cause:

- IS = invalid operand due to stack overflow/underflow
- I = invalid operand due to other cause
- D = denormal operand
- Z = zero-divide
- O = Overflow
- U = Underflow
- P = Inexact result (precision)

F2XM1 Compute 2^X-1

Exceptions: P, U, D, I, IS

F2XM1 (no operands)

Operands		Execution clocks				
		87	287	387	486	586
D9 F0	F2XM1		211-476	211-476	242(140-279)	13-57

FABS Absolute value

Exceptions: IS

FABS (no operands)

Operands		Execution clocks				
	87	287	387	486		
No operands	10-17	10-17	22	3	2	FABS

FADD Add real

Exceptions: I, D, O, U, P, IS

FADD destination, source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
ST,ST(i)	70-100	70-100	23-34	10(8-20)	2	FADD ST,ST(4)
ST(i),ST						FADD ST(2), ST
short real	90-120+EA	90-120	24-32	10(8-20)	2-4	FADD AIR_TEMP[SI]
long real	95-125+EA	95-125	29-37	10(8-20)	2-4	FADD [BX].MEAN

FADDP Add real and pop

Exceptions: I, D, O, U, P, IS

FADDP destination, source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
ST(i),ST	75-105	75-105	23-34	10(8-20)	2	FADDP ST(2),ST

FBLD

Packed decimal (BCD) load

Exceptions: I

FBLD source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
Packed decimal	290-310	290-310	5	75(70-103)	2-4	FBLD YTD_SALES

FBSTP

Packed decimal (BCD) store and pop

Exceptions: I

FBSTP destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
Packed decimal	520-540+EA	520-540+EA	512-534	175(172-176)	2-4	FBSTP [BX].FORECAST

FCHS

Change sign

Exceptions: I

FCHS (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	10-17	10-17	24-25	6	2	FCHS

FCLEX FNCLEX

Clear exceptions

Exceptions: None

FCLEX/FNCLEX (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	2-8	2-8	11	7		FNCLEX

FCOM

Compare real

Exceptions: I, D

FCOM //source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
//ST(i)	40-50	40-50	24	4		FCOM ST(1)
short real	60-70+EA	60-70	26	4	2-4	FCOM [BP].UPPER_LIMIT
long real	65-75+EA	65-75	31	4	2-4	FCOM WAVELENGTH

FCOMP

Compare real and pop

Exceptions: I, D

FCOMP //source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
//ST(i)	42-52	45-52	26	4	2	FCOMP ST(2)
short real	63-73+EA	63-73	26	4	2-4	FCOMP [BP+2].N_READINGS
long real	67-77+EA	67-77	31	4	2-4	FCOMP DENSITY

FCOMPP

Compare real and pop twice

Exceptions: I, D

FCOMPP (no operands)

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
No operands	45-55	45-55	26	5	2	FCOMPP

FCOS

Cosine of ST(0)
80387 and greater

Exceptions: IS, I, D, U, P

FCOS

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
No operands			123-772*	241(193-279)	2	FCOS

*These timings hold for operands in the range *|x| /4*. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

FDECSTP

Decrement stack pointer

Exceptions: None

FDECSTP (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	6-12	6-12	22	3	2	FDECSTP

FDISI FNDISI

Disable interrupts
8087 only

Exceptions: None

FDISI (no operands)

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Example
No operands	5	2-8	0	2	FDISI

FDIV

Divide real

Exceptions: I, D, Z, O, U, P

FDIV //source/destination, source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
//ST(i),ST	193-203	193-203	88-91	73	2	FDIV
short real	215-225	215-225	89	73	2-4	FDIV DISTANCE
long real	220-230	220-230	94	73	2-4	FDIV ARC[DI]
//ST,ST(i)				73		

FDIVP

Divide real and pop

Exceptions: I, D, Z, O, U, P

FDIVP destination, source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
//ST(i),ST	197-207	198-209	88-91	73	2	FDIVP ST(4),ST

FDIVR

Divide real reversed

Exceptions: I, D, Z, O, U, P

FDIVR //source/destination, source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
//ST,ST(i)/	194-204	198-208	88-91	73	2	FDIVR ST(2),ST
ST(i),ST				73		
short real	216-226+EA	215-225	89	73	2-4	FDIVR [BX].PULSE_RATE
long real	221-231+EA	220-230	94	73	2-4	FDIVR RECORDER.FREQUENCY

FDIVRP

Divide real reversed and pop

Exceptions: I, D, Z, O, U, P

FDIVRP destination, source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
ST(i),ST	198-208	198-208	88-91	73	2	FDIVRP ST(1),ST

FENI
FNENI

Enable interrupts
8087 only

Exceptions: None

FENI (no operands)

Operands	Execution clock	Code bytes	Example
	87		
(no operands)	5(2-8)	2	FNENI

FFREE

Free register

Exceptions: None

FFREE destination

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
ST(i)	9-16	9-16	18	3	2	FFREE ST(1)

FIADD

Integer add

Exceptions: I, D, O, P

FIADD source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	102-137+EA	102-137	71-85	22.5(19-32)	2-4	FIADD DISTANCE_TRAVELLED
short integer	108-143+EA	108-143	57-72	24(20-35)	2-4	FIADD PULSE_COUNT [SI]

FICOM

Integer compare

Exceptions: I, D

FICOM source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	72-86+EA	72-86	71-75	18(16-20)	2-4	FICOM TOOL.N_PASSES
short integer	78-91+EA	78-91	56-63	16.5(15-17)	2-4	FICOM [BP+4].PARM_COUNT

FICOMP

Integer compare and pop

Exceptions: I, D

FICOMP source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	74-88+EA	74-88	71-75	18(16-20)	2-4	FICOMP [BP].LIMIT [SI]
short integer	80-93+EA	80-93	56-63	16.5(15-17)	2-4	FICOMP N_SAMPLES

FIDIV

Integer divide

Exceptions: I, D, Z, O, U, P

FIDIV source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	224-238+EA	224-238	136-140	73	2-4	FIDIV SURVEY.OBSERVATIONS
short integer	230-243+EA	230-243	120-127	73	2-4	FIDIV RELATIVE_ANGLE [DI]

FIDIVR Integer divide reversed

Exceptions: I, D, Z, O, U, P

FIDIVR source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	225-239+EA	224-238	135-141	73	2-4	FIDIVR [BP].X_COORD
short integer	231-245+EA	230-243	121-128	73	2-4	FIDIVR FREQUENCY

FILD Integer load

Exceptions: I

FILD source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	46-54+EA	46-54	61-65	11.5(9-12)	2-4	FILD [BX].SEQUENCE
short integer	52-60+EA	52-60	45-52	14.5(13-16)	2-4	FILD STANDOFF [DI]
long integer	60-68+EA	60-68	56-67	16.8(10-18)	2-4	FILD RESPONSE.COUNT

FIMUL Integer multiply

Exceptions: I, D, O, P

FIMUL source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	124-138+EA	124-138	76-87	8	2-4	FIMUL BEARING
short integer	130-144+EA	130-144	61-82	8	2-4	FIMUL POSITION.Z_AXIS

FINCSTP Increment stack pointer

Exceptions: None

FINCSTP (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	6-12	6-12	21	3	2	FINCSTP

FINIT FNINIT

Initialize processor

Exceptions: None

FINIT/FNINIT (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	2-8	2-8	33	17	2	FINIT

FIST

Integer store

Exceptions: I, P

FIST destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	80-90+EA	80-90	82-95	33.4(29-34)	2-4	FIST OBS.COUNT [SI]
short integer	82-92+EA	82-92	79-93	32.4(28-34)	2-4	FIST [BP;].FACTORED_PULSES

FISTP

Integer store and pop

Exceptions: I, P

FISTP destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	82-92+EA	82-92	82-95	33.4(29-34)	2-4	FISTP [BX]. ALPHA_COUNT [SI]
short integer	84-94+EA	84-94	79-93	33.4(29-34)	2-4	FISTP CORRECTED_TIME
long integer	94-105+EA	94-105	80-97	33.4(29-34)	2-4	FISTP PANEL. N_READINGS

FISUB

Integer subtract

Exceptions: I, D, O, P

FISUB source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
word integer	102-137+EA	102-137	71-83	22.5(19-32)	2-4	FISUB BASE_FREQUENCY
short integer	108-143+EA	108-143	57-82	24(20-35)	2-4	FISUB TRAIN_SIZE [DI]

FISUBR

Integer subtract reversed

Exceptions: I, D, O, P

FISUBR source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
word integer	103-139+EA	102-137	72-84	22.5(19-32)	2-4	FISUBR FLOOR [BX][SI]
short integer	109-144+EA	108-143	58-83	24(20-35)	2-4	FISUBR BALANCE

FLD

Load real

Exceptions: I, D

FLD source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
ST(i)	17-22	17-22	14	4	2	FLD ST(0)
short real	38-56+EA	38-56	20	3	2-4	FLD READING [SI].PRESSURE
long real	40-60+EA	40-60	25	3	2-4	FLD [BP].TEMPERATURE
Temp real	53-65+EA	53-65	44	6	2-4	FLD SAVEREADING

FLDCW

Load control word

Exceptions: None

FLDCW source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
2 bytes	7-14+EA	7-14	19	4	2-4	FLDCW CONTROL_WORD

FLDENV

Load environment

Exceptions: None

FLDENV source

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
14 bytes	35-45+EA	35-45	71	44 real or virtual 34 protected	2-4	FLDENV [BP+6]

FLDLG2 Load $\log_{10}2$

Exceptions: I

FLDLG2 (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	18-24	18-24	41	8	2	FLDLG2

FLDLN2 Load $\log_e 2$

Exceptions: I

FLDLN2 (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	17-23	17-23	41	8	2	FLDLN2

FLDL2E Load $\log_2 e$

Exceptions: I

FLDL2E (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	15-21	15-21	40	8	2	FLDL2E

FLDL2T Load $\log_2 10$

Exceptions: I

FLDL2T (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	16-22	16-22	40	8	2	FLDL2T

FLDPI

Load P (pi)

Exceptions: I

FLDPI (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	16-22	16-22	40	8	2	FLDPI

FLDZ

Load +0.0

Exceptions: I

FLDZ (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	11-17	11-17	20	4	2	FLDZ

FLD1

Load +1.0

Exceptions: I

FLD1 (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	15-21	15-21	24	4	2	FLD1

FMUL

Multiply real

Exceptions: I, D, O, U, P

FMUL //source/destination,source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
//ST(i),ST/ST, 90-105,ST(1)*	90-105	90-145	29-57	16	2	FMUL ST,ST(3)
//ST(i),ST/ST, ST,ST(1)	130-145	90-145	29-57	16	2	FMUL ST,ST(3)
short real	110-125+EA	110-125	27-35	11	2-4	FMUL SPEED_FACTOR
long real*	112-126+EA	112-168	32-57		2-4	FMUL [BP].HEIGHT
long real	154-168+EA	112-168	32-57	14	2-4	FMUL [BP].HEIGHT

*Occurs when one or both operands is "short"--it has 40 trailing zeros in its fraction (for example, it was loaded from a short-real memory operand).

FMULP

Multiply real and pop

Exceptions: I, D, O, U, P

FMULP destination,source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
ST(i),ST*	94-108	198-208	29-57		2	FMULP ST(1),ST
ST(i),ST	134-148	198-208	29-57	16	2	FMULP ST(1),ST

*Occurs when one or both operands is "short"--it has 40 trailing zeros in its fraction (for example, it was loaded from a short-real memory operand).

FNOP

No operation

Exceptions: None

FNOP (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	10-16	10-16	12	3	2	FNOP

FPATAN

Partial arctangent

Exceptions: U, P (operands not checked)

FPATAN (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	250-800	250-800	314-487	5(2-17)	2	FPATAN

FPREM

Partial remainder

Exceptions: I, D, U

FPREM (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	15-190	15-190	74-155	2(2-8)	2	FPREM

FPREM1

Partial remainder
80387 and greater

Exceptions: I, D, U

FPREM (no operands)

Operands	87	287	387	486	Code bytes	Example
No operands			95-185	94.5(72-167)	2	FPREM1

FPTAN

Partial tangent

Exceptions: I, P (operands not checked)

FPTAN (no operands)

Operands	87	287	387	486	Code bytes	Example
No operands	30-540	30-540	191-573	244(200-273)	2	FPTAN

FRNDINT

Round to integer

Exceptions: I, P

FRNDINT (no operands)

Operands	87	287	387	486	Code bytes	Example
No operands	16-50	16-50	66-80	29.1(21-30)	2	FRNDINT

FRSTOR

Restore saved state

Exceptions: None

FRSTOR source

Operands	87	287	387	486	Code bytes	Example
94 bytes	197-207+EA	205-215	308	131 real or virtual 120 protected	2-4	FRSTOR [BP]

Note: The 80287 execution clock count for this instruction is not meaningful in determining overall instruction execution time. For typical frequency ratios of the 80286 and 80287 clocks, 80287 execution occurs in parallel with the operand transfers. The operand transfers determine the overall execution time of the instructions. For 80286:80287 clock frequency ratios of 4:8, 1:1, and 8:5, the overall execution clock count for this instruction is estimated at 490, 302, and 227 80287 clocks, respectively.

FSAVE FNSAVE

Save state

Exceptions: None

FSAVE/FNSAVE destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
94 bytes	197-207+EA	205-215	375-376		2-4	FSAVE [BP]

Note: The 80287 execution clock count for this instruction is not meaningful in determining overall instruction execution time. For typical frequency ratios of the 80286 and 80287 clocks, 80287 execution occurs in parallel with the operand transfers. The operand transfers determine the overall execution time of the instruction. For 80286:80287 clock frequency ratios of 4:8, 1:1, and 8:5, the overall execution clock count for this instruction is estimated at 376, 233, and 174 80287 clocks, respectively.

FSCALE

Scale

Exceptions: I, O, U

FSCALE (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	32-38	32-38	67-86	31(30-32)	2	FSCALE

FSETPM

Set protected mode

Exceptions: None

FSETPM (no operands)

Operands	Execution clock	Code bytes	Example
	287		
No operands	2-8	2	FSETPM

FSIN

Sine of ST(0) 80387 and greater

Exceptions: IS, I, D, U, P

FSIN

Operands	Execution clocks		Code bytes	Example
	387	486		
No operands	122-771*	241(193-279)	2	FSIN

*These timings hold for operands in the range *|x|* /4. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

FSINCOS

Sine and cosine of ST(0)
80387 and greater

Exceptions: IS, I, D, U, P

FSINCOS

Operands	Execution clocks		Code bytes	Example
	387	486		
No operands	194-809*	291(243-329)	2	FSINCOS

*These timings hold for operands in the range |x| /4. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

FSQRT

Square root

Exceptions: I, D, P

FSQRT (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	180-186	180-186	122-129	85.5(83-87)	2	FSQRT

FST

Store real

Exceptions: I, O, U, P

FST destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
ST(i)	15-22	15-22	11	3	2	FST ST(3)
short real	84-90+EA	84-90	44	7	2-4	FST CORRELATION [DI]
long real	96-104+EA	96-104	45	8	2-4	FST MEAN_READING

FSTCW
FNSTCW

Store control word

Exceptions: None

FSTCW destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
2 bytes	12-18+EA	12-18	15		2-4	FSTCW SAVE_CONTROL

FSTENV FNSTENV

Store environment

Exceptions: None

FSTENV destination

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
14 bytes	40-50+EA	40-50	103-104		2-4	FSTENV [BP]

FSTP

Store real and pop

Exceptions: I, O, U, P

FSTP destination

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
ST(i)	17-24	17-24	12	3	2	FSTP ST(2)
short real	86-92+EA	86-92	44	7	2-4	FSTP [BX]. ADJUSTED_RPM
long real	98-106+EA	98-106	45	8	2-4	FSTP TOTAL_DOSAGE
Temp real	52-58+EA	52-58	53	6	2-4	FSTP REG_SAVE [SI]

FSTSW FNSTSW

Store status word

Exceptions: None

FSTSW/FNSTSW destination

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
2 bytes	12-18+EA	12-18	15	3	2-4	FSTSW SAVE_STATUS

FSTSW AX FNSTSW AX

Store status word to AX

Exceptions: None

FSTSW destination

Operands		Execution clocks			Code bytes	Example
	87	287	387	486		
AX		10-16	13	3	2	FSTSW AX

FSUB Subtract real

Exceptions: I, D, O, U, P

FSUB //source/destination,source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
//ST,ST/(i)/ ST(i),ST	70-100	70-100	26-37	7(5-17)	2	FSUB ST,ST(2)
short real	90-120+EA	90-120	24-32	7(5-17)	2-4	FSUB BASE_VALUE
long real	95-125+EA	95-125	28-36	7(5-17)	2-4	FSUB COORDINATE.X

FSUBP Subtract real and pop

Exceptions: I, D, O, U, P

FSUBP destination, source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
ST(i),ST	75-105	75-105	26-37	7(5-17)	2	FSUBP ST(2),ST

FSUBR Subtract real reversed

Exceptions: I, D, O, U, P

FSUBR //source/destination, source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
//ST,ST(i)/ ST(i),ST	70-100	70-100	26-37	7(5-17)	2	FSUBR ST,ST(1)
short real	90-120+EA	90-120	25-33	7(5-17)	2-4	FSUBR VECTOR [SI]
long real	95-125+EA	95-125	29-37	7(5-17)	2-4	FSUBR [BX].INDEX

FSUBRP Subtract real reversed and pop

Exceptions: I, D, O, U, P

FSUBRP destination, source

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
ST(i),ST	75-105	75-105	26-37	7(5-17)	2	FSUBRP ST(1),ST

FTST

Test stack top against +0.0

Exceptions: I, D

FTST (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	38-48	38-48	28	4	2	FTST

FUCOM

Unordered compare
80387 and greater

Exceptions: IS, I, D

Operands	Execution clocks		Code bytes	Example
	387	486		
//ST(i)	24	4	2	FUCOM ST(1)

FUCOMP

Unordered compare
80387 and greater

Exceptions: IS, I, D

Operands	Execution clocks		Code bytes	Example
	387	486		
//ST(i)	26	4	2	FUCOMP ST(2)

FUCOMPP

Unordered compare
80387 and greater

Exceptions: IS, I, D

Operands	Execution clocks		Code bytes	Example
	387	486		
No operands	26	5	2	FUCOMPP

FWAIT

Wait

Exceptions: None (CPU instruction)

FWAIT (no operands)

Operands	Execution clocks		Code bytes	Example
	387	486		
No operands	3+5*n**	1-3	1	FWAIT

**n* = number of time CPU examines BUSY line before 80287 completes execution of previous instruction.

FXAM

Examine stack top

Exceptions: None

FXAM (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	12-23	12-23	30-38	8	2	FXAM

FXCH

Exchange registers

Exceptions: I

FXCH //destination

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
//ST(i)	10-15	10-15	18	4	2	FXCH ST(2)

FXTRACT

Extract exponent and significant

Exceptions: I

FXTRACT (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	27-55	27-55	70-76	19(16-20)	2	FXTRACT

FYL2X

Y * $\log_2 X$

Exceptions: P (operands not checked)

FYL2X (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	900-1100	900-1100	120-538	311(196-329)	2	FYL2X

FYL2XP1

$Y * \log_2(X+1)$

Exceptions: P (operands not checked)

FYL2XP1 (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	700-1000	700-1000	257-547	313(171-326)	2	FYL2XP1

F2XM1

2^X-1

Exceptions: U, P (operands not checked)

F2XM1 (no operands)

Operands	Execution clocks				Code bytes	Example
	87	287	387	486		
No operands	310-630	310-630	211-476	242(140-279)	2	F2XM1

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000.
Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # LSM1240WW21771 • BOR 6285

BORLAND • USER FRIENDLY • EARTH FRIENDLY •