

Intel Pentium Instruction Set Reference

Produced by mindweaver@technologist.com

This is a complete reference of all Intel Pentium instructions, with the exception of MMX and FPU instructions.

This document has been painstakingly rendered into a nice searchable Windows help file for you to enjoy. Various source documents have been used to produce this. The official Intel Pentium Instruction Set Reference Manual for the mnemonics, opcodes, descriptions and flags; and Quantasm's instruction set reference for the opcode timing, size, and pairing data.

You can also check out the [document conventions](#) and [instruction page description](#).

Not part of the instruction set information, but useful nonetheless is the [basic architecture overview](#).

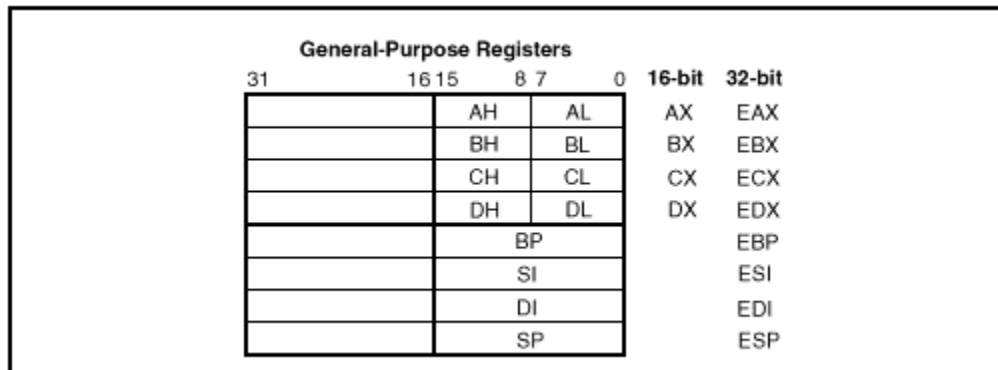
Enjoy

Version History

1.0	15-04-99	First Version.
-----	----------	----------------

Basic Architecture Overview

The following diagram shows the general purpose registers:

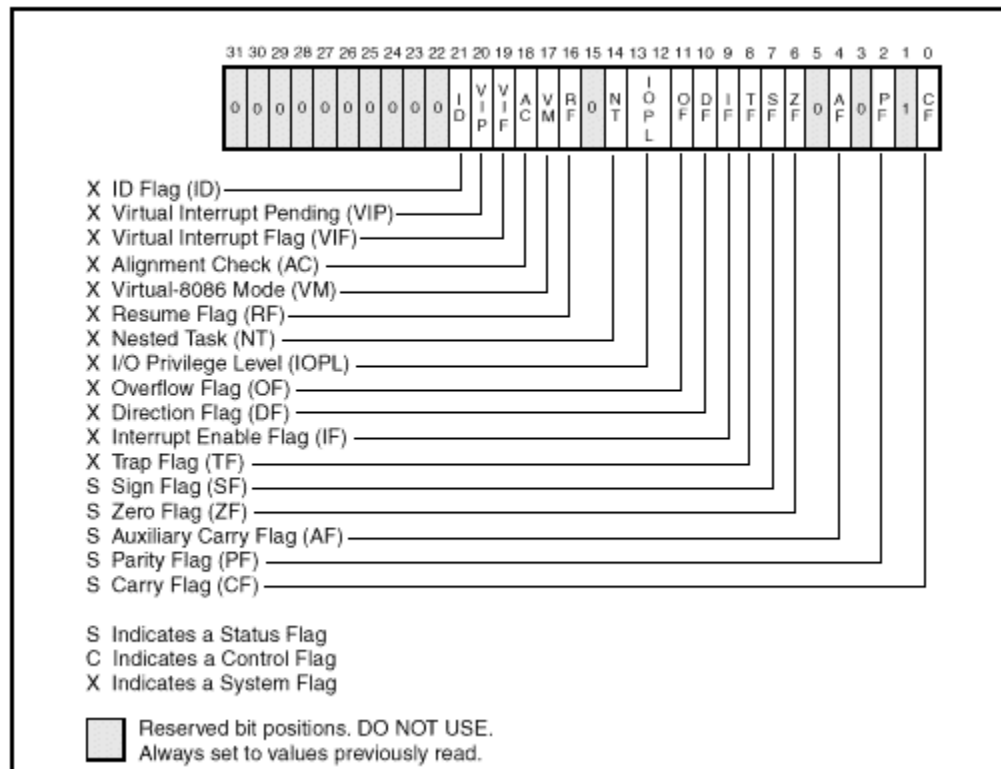


EAX	Accumulator for operands and results data.
EBX	Pointer to data in the DS segment.
ECX	Counter for string and loop operations.
EDX	I/O pointer.
EBP	Pointer to data on the stack (in the SS segment).
ESI	Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
EDI	Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
ESP	Stack pointer (in the SS segment).

The 16-Bit Segment Registers are: -

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Data Segment
FS	Data Segment
GS	Data Segment

The format of the EFLAGS register: -



Mnemonic - Description

Opcodes	Official Operands	Description
---------	-------------------	-------------

Description

Long description.

Operands

Generalised operands	Bytes Size	Clocks Clock cycles	Pairing info
----------------------	----------------------	-------------------------------	--------------

Flags

Affected Flags

Opcodes are in Hex.

Conventions

The following conventions are used throughout this document.

Official Operands

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 and rel32	A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
ptr16:16 and 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 left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
r8	One of the byte general-purpose registers AL, CL, DL, BL, AH, CH, DH, or BH.
r16	One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI.
r32	One of the doubleword general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.
imm8	An immediate byte value. The imm8 symbol 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	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 inclusive.
r/m8	A byte operand that is either the contents of a byte general-purpose register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.
r/m16	A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, BX, CX, DX, SP, BP, SI, and DI. The contents of memory are found at the address provided by the effective address computation.
r/m32	A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The contents of memory are found at the address provided by the effective address computation.
m	A 16- or 32-bit operand in memory.
m8	A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions and the XLAT instruction.
m16	A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only

	with the string instructions.
m32	A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
m64	A memory quadword operand in memory. This nomenclature is used only with the CMPXCHG8B instruction.
m16:16, m16:32	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, m32&32	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. The 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. The m16&32 operand 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 GDTR and IDTR registers.
moffs8, moffs16, moffs32	A simple memory variable (memory offset) of type byte, word, or doubleword 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, and GS=5.
m32real, m64real, m80real	A single-, double-, and extended-real (respectively) floating-point operand in memory.
m16int, m32int, m64int	A word-, short-, and long-integer (respectively) floating-point operand in memory.
ST or ST(0)	The top element of the FPU register stack.
ST(i)	The i th element from the top of the FPU register stack. (i = 0 through 7)
mm	An MMX™ register. The 64-bit MMX registers are: MM0 through MM7.
mm/m32	The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
mm/m64	An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

Generalised Operands

acc	AL, AX or EAX unless specified otherwise
reg	any general register
r8	any 8-bit register
r16	any general purpose 16-bit register
r32	any general purpose 32-bit register
imm	immediate data
imm8	8-bit immediate data
imm16	16-bit immediate data
mem	memory address
mem8	address of 8-bit data item
mem16	address of 16-bit data item
mem32	address of 32-bit data item
mem48	address of 48-bit data item
dest	16/32-bit destination
short	8-bit destination

Instruction Size

The byte count includes the opcode length and length of any required displacement or immediate data. If the displacement is optional, it is shown as d() with the possible lengths in parentheses. If the immediate data is optional, it is shown as i() with the

possible lengths in parentheses.

Pairing Categories

NP = not pairable

UV = pairable in the U pipe or V pipe

PU = pairable in the U pipe only

PV = pairable in the V pipe only

Pentium Optimisations

By mindweaver@technologist.com

Instruction Timing

To time the number of clock cycles that a block of code takes you can use the [rdtsc](#) instruction. See it's entry for full information.

The following code ultimately sets **eax** to the number of clock cycles taken by the *usercode*: -

```
rdtsc          // get CPU clocks
mov clocks,eax // save clock counter

[usercode]

rdtsc          // get CPU clocks
mov ecx, clocks // get the old clock counter
sub eax, ecx   // calc clock cycles taken
sub eax, $0E   // subtract clocks wasted on rdtsc's
```

Where *clocks* is a 32-Bit variable.

If you are using assembly embedded in a high level language your assembler might not support the **rdtsc** mnemonic, as Borland Delphi 3 doesn't. In this case you can use the following construct in place of the **rdtsc** instruction: -

```
dw $310F      // two bytes in hex
```

Note that if the code being timed is quite long, the time value might not be accurate due to interrupts occurring during your code execution.

AAA - Ascii Adjust for Addition

37 AAA ASCII adjust AL after addition

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register is incremented by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are cleared to 0.

Operands	Bytes	Clocks
	1	3 NP

Flags

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

AAD - Ascii Adjust AX Before Division

D5 0A	AAD	ASCII adjust AX before division
D5 1b	(No mnemonic)	Adjust AX before division to number base imm8

Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to $(AL + (10 * AH))$, and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the "Operation" section below), by setting the imm8 byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 imm8).

Operands	Bytes	Clocks
	2	10 NP

Flags

The SF, ZF, and PF flags are set according to the result; the OF, AF, and CF flags are undefined.

AAM - Ascii Adjust AX After Multiply

D4 0A	AAM	ASCII adjust AX after multiply
D4 ib	(No mnemonic)	Adjust AX after multiply to number base imm8

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the imm8 byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 imm8).

Operands	Bytes	Clocks
	2	18 NP

Flags

The SF, ZF, and PF flags are set according to the result. The OF, AF, and CF flags are undefined.

AAS - ASCII Adjust AL After Subtraction

3F AAS ASCII adjust AL after subtraction

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register is decremented by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top nibble set to 0.

Operands	Bytes	Clocks
	1	3 NP

Flags

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

ADC - Add with Carry

14 ib	ADC AL, imm8	Add with carry imm8 to AL
15 iw	ADC AX, imm16	Add with carry imm16 to AX
15 id	ADC EAX, imm32	Add with carry imm32 to EAX
80 /2 ib	ADC r/m8, imm8	Add with carry imm8 to r/m8
81 /2 iw	ADC r/m16, imm16	Add with carry imm16 to r/m16
81 /2 id	ADC r/m32, imm32	Add with CF imm32 to r/m32
83 /2 ib	ADC r/m16, imm8	Add with CF sign-extended imm8 to r/m16
83 /2 id	ADC r/m32, imm8	Add with CF sign-extended imm8 into r/m32
10 / r	ADC r/m8, r8	Add with carry byte register to r/m8
11 / r	ADC r/m16, r16	Add with carry r16 to r/m16
11 / r	ADC r/m32, r32	Add with CF r32 to r/m32
12 / r	ADC r8, r/m8	Add with carry r/m8 to byte register
13 / r	ADC r16, r/m16	Add with carry r/m16 to r16
13 / r	ADC r32, r/m32	Add with CF r/m32 to r32

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

Operands	Bytes	Clocks	
reg, reg	2	1	PU
mem, reg	2+d(0,2)	3	PU
reg, mem	2+d(0,2)	2	PU
reg, imm	2+i(1,2)	1	PU
mem, imm	2+d(0,2)+i(1,2)	3	PU*
acc, imm	1+i(1,2)	1	PU

* = not pairable if there is a displacement and immediate

Flags

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

ADD - Add

04 ib	ADD AL, imm8	Add imm8 to AL
05 iw	ADD AX, imm16	Add imm16 to AX
05 id	ADD EAX, imm32	Add imm32 to EAX
80 /0 ib	ADD r/m8, imm8	Add imm8 to r/m8
81 /0 iw	ADD r/m16, imm16	Add imm16 to r/m16
81 /0 id	ADD r/m32, imm32	Add imm32 to r/m32
83 /0 ib	ADD r/m16, imm8	Add sign-extended imm8 to r/m16
83 /0 id	ADD r/m32, imm8	Add sign-extended imm8 to r/m32
00 / r	ADD r/m8, r8	Add r8 to r/m8
01 / r	ADD r/m16, r16	Add r16 to r/m16
01 / r	ADD r/m32, r32	Add r32 to r/m32
02 / r	ADD r8, r/m8	Add r/m8 to r8
03 / r	ADD r16, r/m16	Add r/m16 to r16
03 / r	ADD r32, r/m32	Add r/m32 to r32

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0,2)	3 UV
reg, mem	2+d(0,2)	2 UV
reg, imm	2+i(1,2)	1 UV
mem, imm	2+d(0,2)+i(1,2)	3 UV*
acc, imm	1+i(1,2)	1 UV

* = not pairable if there is a displacement and immediate

Flags

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

AND - Logical AND

24 ib	AND AL, imm8 AL	AND imm8
25 iw	AND AX, imm16 AX	AND imm16
25 id	AND EAX, imm32 EAX	AND imm32
80 /4 ib	AND r/m8, imm8 r/m8	AND imm8
81 /4 iw	AND r/m16, imm16 r/m16	AND imm16
81 /4 id	AND r/m32, imm32 r/m32	AND imm32
83 /4 ib	AND r/m16, imm8 r/m16	AND imm8 (sign-extended)
83 /4 id	AND r/m32, imm8 r/m32	AND imm8 (sign-extended)
20 /r	AND r/m8, r8 r/m8	AND r8
21 /r	AND r/m16, r16 r/m16	AND r16
21 /r	AND r/m32, r32 r/m32	AND r32
22 /r	AND r8, r/m8 r8	AND r/m8
23 /r	AND r16, r/m16 r16	AND r/m16
23 /r	AND r32, r/m32 r32	AND r/m32

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) 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.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0,2)	3 UV
reg, mem	2+d(0,2)	2 UV
reg, imm	2+i(1,2)	1 UV
mem, imm	2+d(0,2)+i(1,2)	3 UV*
acc, imm	1+i(1,2)	1 UV

* = not pairable if there is a displacement and immediate

Flags

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

ARPL - Adjust RPL Field of Segment Selector

63 / r ARPL r/m16,r16

Adjust RPL of r/m16 to not less than RPL of r16

Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program. (The segment selector for the application program's code segment can be read from the stack following a procedure call.)

Operands	Bytes	Clocks
reg, reg	2	7 NP
mem, reg	2+d(0-2)	7 NP

Flags

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, is cleared to 0.

BOUND - Check Array Index Against Bounds

62 / r	BOUND r16,m16&16	Check if r16 (array index) is within bounds specified by m16&16
62 / r	BOUND r32,m32&32	Check if r32 (array index) is within bounds specified by m16&16

Description

Determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. (When a this exception is generated, the saved return instruction pointer points to the BOUND instruction.)

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

Operands	Bytes	Clocks
reg, mem	4	8 NP

Flags

None.

BSF - Bit Scan Forward

0F BC	BSF r16,r/m16	Bit scan forward on r/m16
0F BC	BSF r32,r/m32	Bit scan forward on r/m32

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the contents source operand are 0, the contents of the destination operand is undefined.

Operands	Bytes	Clocks
r16, r16	3	6-34 NP
r32, r32	3	6-42 NP
r16, m16	3+d(0,1,2)	6-35 NP
r32, m32	3+d(0,1,2,4)	6-43 NP

Flags

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

BSR - Bit Scan Reverse

0F BD	BSR r16,r/m16	Bit scan reverse on r/m16
0F BD	BSR r32,r/m32	Bit scan reverse on r/m32

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the contents source operand are 0, the contents of the destination operand is undefined.

Operands	Bytes	Clocks
r16, r16	3	7-39 NP
r32, r32	3	7-71 NP
r16, m16	3+d(0,1,2)	7-40 NP
r32, m32	3+d(0,1,2,4)	7-72 NP

Flags

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

BSWAP - Byte Swap

0F C8+ rd

BSWAP r32

Reverses the byte order of a 32-bit register.

Description

Reverses the byte order of a 32-bit (destination) register: bits 0 through 7 are swapped with bits 24 through 31, and bits 8 through 15 are swapped with bits 16 through 23. This instruction is provided for converting little-endian values to big-endian format and vice versa.

To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

Operands

r32

Bytes

2

Clocks

1 NP

Flags

None.

BT - Bit Test

OF A3	BT r/m16,r16	Store selected bit in CF flag
OF A3	BT r/m32,r32	Store selected bit in CF flag
OF BA /4 ib	BT r/m16,imm8	Store selected bit in CF flag
OF BA /4 ib	BT r/m32,imm8	Store selected bit in CF flag

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

Effective Address + $(4 * (\text{BitOffset} \text{ DIV } 32))$

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

Effective Address + $(2 * (\text{BitOffset} \text{ DIV } 16))$

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

Operands	Bytes	Clocks
reg, reg	3	4 NP
mem, reg	$3+d(0,1,2,4)$	9 NP
reg, imm8	$3+i(1)$	4 NP
mem, imm8	$3+d(0,1,2,4)+i(1)$	4 NP

Flags

The CF flag contains the value of the selected bit. The OF, SF, ZF, AF, and PF flags are undefined.

BTC - Bit Test and Compliment

OF BB	BTC r/m16,r16	Store selected bit in CF flag and complement
OF BB	BTC r/m32,r32	Store selected bit in CF flag and complement
OF BA /7 ib	BTC r/m16,imm8	Store selected bit in CF flag and complement
OF BA /7 ib	BTC r/m32,imm8	Store selected bit in CF flag and complement

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "[BT—Bit Test](#)" for more information on this addressing mechanism.

Operands	Bytes	Clocks
reg, reg	3	7 NP
mem, reg	3+d(0,1,2,4)	13 NP
reg, imm8	3+i(1)	7 NP
mem, imm8	3+d(0,1,2,4)+i(1)	8 NP

Flags

The CF flag contains the value of the selected bit before it is complemented. The OF, SF, ZF, AF, and PF flags are undefined.

BTR - Bit Test and Reset

0F B3	BTR r/m16,r16	Store selected bit in CF flag and clear
0F B3	BTR r/m32,r32	Store selected bit in CF flag and clear
0F BA /6 ib	BTR r/m16,imm8	Store selected bit in CF flag and clear
0F BA /6 ib	BTR r/m32,imm8	Store selected bit in CF flag and clear

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "[BT—Bit Test](#)" in this chapter for more information on this addressing mechanism.

Operands	Bytes	Clocks
reg, reg	3	7 NP
mem, reg	3+d(0,1,2,4)	13 NP
reg, imm8	3+i(1)	7 NP
mem, imm8	3+d(0,1,2,4)+i(1)	8 NP

Flags

The CF flag contains the value of the selected bit before it is cleared. The OF, SF, ZF, AF, and PF flags are undefined.

BTS - Bit Test and Set

0F AB	BTS r/m16,r16	Store selected bit in CF flag and set
0F AB	BTS r/m32,r32	Store selected bit in CF flag and set
0F BA /5 ib	BTS r/m16,imm8	Store selected bit in CF flag and set
0F BA /5 ib	BTS r/m32,imm8	Store selected bit in CF flag and set

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively (see Figure 3-1). If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string (see Figure 3-2). The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "[BT—Bit Test](#)" for more information on this addressing mechanism.

Operands	Bytes	Clocks
reg, reg	3	7 NP
mem, reg	3+d(0,1,2,4)	13 NP
reg, imm8	3+i(1)	7 NP
mem, imm8	3+d(0,1,2,4)+i(1)	8 NP

Flags

The CF flag contains the value of the selected bit before it is set. The OF, SF, ZF, AF, and PF flags are undefined.

CALL - Call Procedure

E8 cw	CALL rel16	Call near, relative, displacement relative to next instruction
E8 cd	CALL rel32	Call near, relative, displacement relative to next instruction
FF /2	CALL r/m16	Call near, absolute indirect, address given in r/m16
FF /2	CALL r/m32	Call near, absolute indirect, address given in r/m32
9A cd	CALL ptr16:16	Call far, absolute, address given in operand
9A cp	CALL ptr16:32	Call far, absolute, address given in operand
FF /3	CALL m16:16	Call far, absolute indirect, address given in m16:16
FF /3	CALL m16:32	Call far, absolute indirect, address given in m16:32

Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call—A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- Task switch—A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 4 of the Intel Architecture Software Developer's Manual, Volume 1, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, Task Management, in the Intel Architecture Software Developer's Manual, Volume 3, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified with the target operand. The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call, an absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16 or r/m32). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. (When accessing an absolute offset indirectly using the stack pointer [ESP] as a base register, the base value used is the value of the ESP before the instruction executes.)

A relative offset (rel16 or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP register. As with absolute offsets, the operand-size attribute determines the size of the target operand (16 or 32 bits).

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers onto the stack for use as a return-instruction pointer. The processor then performs a "far branch" to the code segment and offset specified with the target operand for the called procedure. Here the target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a

memory location (m16:16 or m16:32). With the pointer method, the segment and offset of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following three types of far calls:

- Far call to the same privilege level.
- Far call to a different privilege level (inter-privilege level call).
- Task switch (far call to another task).

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register.

Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. Here again, the target operand can specify the call gate segment selector either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.) On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an (optional) set of parameters from the calling procedure's stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction, is somewhat similar to executing a call through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset in the target operand is ignored.) The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into EIP register so that the task begins executing again at this next instruction.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6, Task Management, in the Intel Architecture Software Developer's Manual, Volume 3, for detailed information on the mechanics of a task switch.

Note that when you execute a task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction, which, because the NT flag is set, will automatically use the previous task link to return to the calling task. (See "Task Linking" in Chapter 6 of the Intel Architecture Software Developer's Manual, Volume 3, for more information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from the JMP instruction which does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, the calls should be made through a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset is saved. Also, the call should be made using a 16-bit call gate so that 16-bit values will be pushed on the stack. See Chapter 16, Mixing 16-Bit and 32-Bit Code, in the Intel Architecture Software Developer's Manual, Volume 3, for more information on making calls between 16-bit and 32-bit code segments.

Operands	Bytes	Clocks
near	3	1 PV
reg	2	2 NP
mem16	2+d(0-2)	2 NP
far	5	4 NP
mem32	2+d(0-2)	4 NP
Protected Mode		
far	5	4-13 NP
mem32	2+d(0-2)	5-14 NP

Cycles not shown for calls through call and task gates

Flags

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

CBW - Convert Byte to Word

98	CBW	AX ← sign-extend of AL
98	CWDE	EAX ← sign-extend of AX

Description

Double the size of the source operand by means of sign extension (see Figure 6-5 in the Intel Architecture Software Developer's Manual, Volume 1). The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.

The CBW and CWDE mnemonics reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16 and the CWDE instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CBW is used and to 32 when CWDE is used. Others may treat these mnemonics as synonyms (CBW/CWDE) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

The CWDE instruction is different from the CWD (convert word to double) instruction. The CWD instruction uses the DX:AX register pair as a destination operand; whereas, the CWDE instruction uses the EAX register as a destination.

Operands	Bytes	Clocks
	1	3 NP

Flags

None.

CWDE - Convert Word to Doubleword

98	CBW	AX ← sign-extend of AL
98	CWDE	EAX ← sign-extend of AX

Description

Double the size of the source operand by means of sign extension (see Figure 6-5 in the Intel Architecture Software Developer's Manual, Volume 1). The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.

The CBW and CWDE mnemonics reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16 and the CWDE instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CBW is used and to 32 when CWDE is used. Others may treat these mnemonics as synonyms (CBW/CWDE) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

The CWDE instruction is different from the CWD (convert word to double) instruction. The CWD instruction uses the DX:AX register pair as a destination operand; whereas, the CWDE instruction uses the EAX register as a destination.

Operands	Bytes	Clocks
	1	3 NP

Flags

None.

CDQ - Convert Double to Quad

99	CWD	DX:AX ← sign-extend of AX
99	CDQ	EDX:EAX ← sign-extend of EAX

Description

Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register (see Figure 6-5 in the Intel Architecture Software Developer's Manual, Volume 1). The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a double-word before doubleword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

Operands	Bytes	Clocks
	1	2 NP

Flags

None.

CWD - Convert Word to Doubleword

99	CWD	DX:AX ← sign-extend of AX
99	CDQ	EDX:EAX ← sign-extend of EAX

Description

Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register (see Figure 6-5 in the Intel Architecture Software Developer's Manual, Volume 1). The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a double-word before doubleword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

Operands	Bytes	Clocks
	1	2 NP

Flags

None.

CLC - Clear Carry Flag

F8 CLC Clear CF flag

Description

Clears the CF flag in the EFLAGS register.

Operands	Bytes	Clocks
	1	2 NP

Flags

The CF flag is cleared to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

CLD - Clear Direction Flag

FC CLD Clear DF flag

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI).

Operands	Bytes	Clocks
	1	2 NP

Flags

The DF flag is cleared to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

CLI - Clear Interrupt Flag

FA CLI Clear interrupt flag; interrupts disabled when interrupt flag cleared

Description

Clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts.

Operands	Bytes	Clocks
	1	7 NP

Flags

The IF is cleared to 0 if the CPL is equal to or less than the IOPL; otherwise, it is not affected. The other flags in the EFLAGS register are unaffected.

CLTS - Clear Task-Switched Flag in CR0

0F 06 CLTS Clears TS flag in CR0

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled "Control Registers" in Chapter 2 of the Intel Architecture Software Developer's Manual, Volume 3, for more information about this flag.

Operands	Bytes	Clocks
	2	10 NP

Flags

The TS flag in CR0 register is cleared.

CMC - Complement Carry Flag

F5 CMC Complement CF flag

Description

Complements the CF flag in the EFLAGS register.

Operands	Bytes	Clocks
	1	2 NP

Flags

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

CMP - Compare Two Operands

3C ib	CMP AL, imm8	Compare imm8 with AL
3D iw	CMP AX, imm16	Compare imm16 with AX
3D id	CMP EAX, imm32	Compare imm32 with EAX
80 /7 ib	CMP r/m8, imm8	Compare imm8 with r/m8
81 /7 iw	CMP r/m16, imm16	Compare imm16 with r/m16
81 /7 id	CMP r/m32, imm32	Compare imm32 with r/m32
83 /7 ib	CMP r/m16, imm8	Compare imm8 with r/m16
83 /7 id	CMP r/m32, imm8	Compare imm8 with r/m32
38 / r	CMP r/m8, r8	Compare r8 with r/m8
39 / r	CMP r/m16, r16	Compare r16 with r/m16
39 / r	CMP r/m32, r32	Compare r32 with r/m32
3A / r	CMP r8, r/m8	Compare r/m8 with r8
3B / r	CMP r16, r/m16	Compare r/m16 with r16
3B / r	CMP r32, r/m32	Compare r/m32 with r32

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (Jcc), condition move (CMOVcc), or SETcc instruction. The condition codes used by the Jcc, CMOVcc, and SETcc instructions are based on the results of a CMP instruction. Appendix B, EFLAGS Condition Codes, in the Intel Architecture Software Developer's Manual, Volume 1, shows the relationship of the status flags and the condition codes.

Operands	Bytes	Clocks	
reg, reg	2	1	UV
mem, reg	2+d(0,2)	2	UV
reg, mem	2+d(0,2)	2	UV
reg, imm	2+i(1,2)	1	UV
mem, imm	2+d(0,2)+i(1,2)	2	UV*
acc, imm	1+i(1,2)	1	UV

* = not pairable if there is a displacement and immediate

Flags

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

CMPS/CMPSB/CMPSW/CMPSD - Compare String Operands

A6	CMPS m8, m8	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPS m16, m16	Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly
A7	CMPS m32, m32	Compares doubleword at address DS:(E)SI with doubleword at address ES:(E)DI and sets the status flags accordingly
A6	CMPSB	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPSW	Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly
A7	CMPD	Compares doubleword at address DS:(E)SI with doubleword at address ES:(E)DI and sets the status flags accordingly

Description

Compares the byte, word, or double word specified with the first source operand with the byte, word, or double word specified with the second source operand and sets the status flags in the EFLAGS register according to the results. Both the source operands are located in memory. The address of the first source operand is read from either the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the second source operand is read from either the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct location. The locations of the source operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI and ES:(E)DI registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), or CMPD (double-word comparison).

After the comparison, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incremented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The CMPS, CMPSB, CMPSW, and CMPD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Variations	Bytes	Clocks
cmpsb	1	5 NP
cmpsw	1	5 NP
cmpsd	1	5 NP
repX cmpsb	2	9+4n NP
repX cmpsw	2	9+4n NP
repX cmpsd	2	9+4n NP

Flags

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

CMPXCHG - Compare and Exchange

0F B0/ r	CMPXCHG r/m8,r8	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
0F B1/ r	CMPXCHG r/m16,r16	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AL.
0F B1/ r	CMPXCHG r/m32,r32	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into AL.

Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Operands	Bytes	Clocks	
reg, reg	3	5	NP
mem, reg	3+d(0-2)	6	NP

Flags

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

CMPXCHG8B - Compare and Exchange 8 Bytes

0F C7 /1 m64 CMPXCHG8B m64 Compare EDX:EAX with m64. If equal, set ZF and load ECX:EBX into m64. Else, clear ZF and load m64 into EDX:EAX.

Description

Compares the 64-bit value in EDX:EAX with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX. The destination operand is an 8-byte memory location. For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Operands	Bytes	Clocks
mem, reg	3+d(0-2)	10 NP

Flags

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.

CPUID - CPU Identification

0F A2 CPUID EAX ← Processor identification information

Description

Provides processor identification information in registers EAX, EBX, ECX, and EDX. This information identifies Intel as the vendor, gives the family, model, and stepping of processor, feature information, and cache information. An input value loaded into the EAX register determines what information is returned.

Information Returned by CPUID Instruction

Initial EAX	Value Information Provided about the Processor	
0	EAX	Maximum CPUID Input Value (2 for the Pentium ®Pro processor and 1 for the Pentium processor and the later versions of Intel486™ processor that support the CPUID instruction).
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
1	EAX	Version Information (Type, Family, Model, and Stepping ID)
	EBX	Reserved
	ECX	Reserved
	EDX	Feature Information
2	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information

The CPUID instruction can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed (see "Serializing Instructions" in Chapter 7 of the Intel Architecture Software Developer's Manual, Volume 3).

When the input value in register EAX is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register (see Table 3-4). A vendor identification string is returned in the EBX, EDX, and ECX registers. For Intel processors, the vendor identification string is "GenuineIntel" as follows:

```
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 *)
```

When the input value is 1, the processor returns version information in the EAX register and feature information in the EDX register (see Figure 3-3).

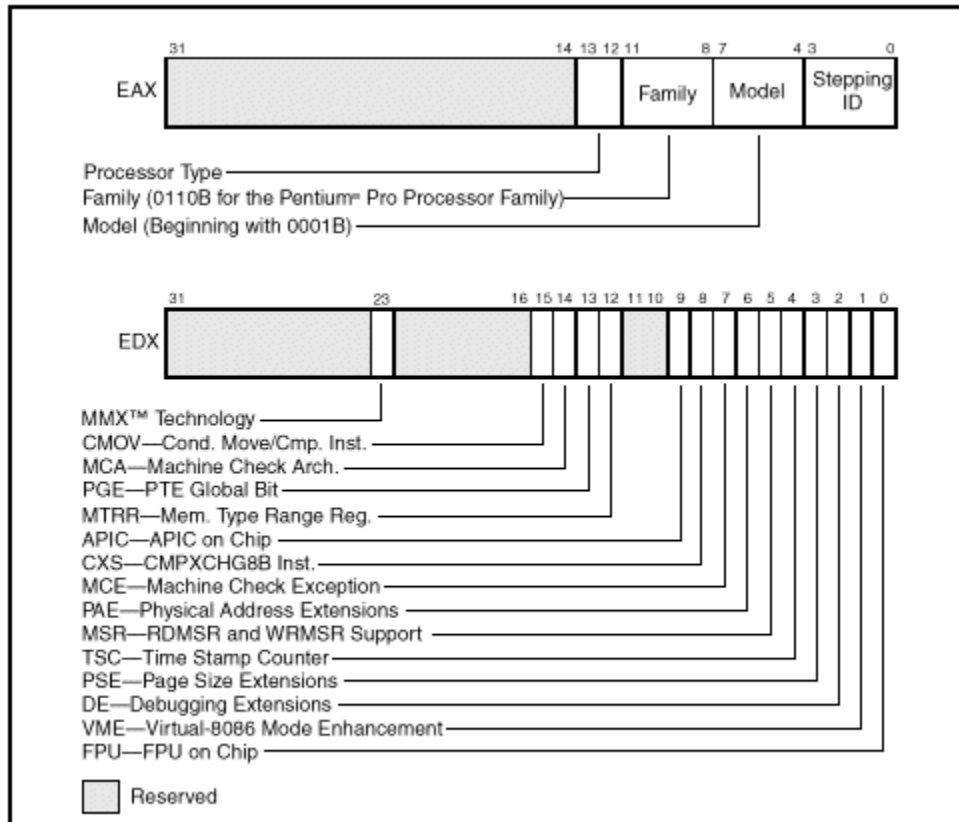


Figure 3-3. Version and Feature Information in Registers EAX and EDX

The version information consists of an Intel Architecture family identifier, a model identifier, a stepping ID, and a processor type. The model, family, and processor type for the first processor in the Intel Pentium Pro family is as follows:

- Model—0001B
- Family—0110B
- Processor Type—00B

See AP-485, Intel Processor Identification and the CPUID Instruction (Order Number 241618), the Intel Pentium® Pro Processor Specification Update (Order Number 242689), and the Intel Pentium® Processor Specification Update (Order Number 242480) for more information on identifying earlier Intel Architecture processors.

The available processor types are given in Table 3-5. Intel releases information on stepping IDs as needed.

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor	*10B
Intel reserved.	11B

Table 3-6 shows the encoding of the feature flags in the EDX register. A feature flag set to 1 indicates the corresponding feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags.

Bit	Feature	Description
0	FPU—Floating-Point Unit	Processor contains an FPU and executes the Intel 387

1	on Chip VME—Virtual-8086 Mode Enhancements	instruction set. Processor supports the following virtual-8086 mode enhancements: <ul style="list-style-type: none"> • CR4.VME bit enables virtual-8086 mode extensions. • CR4.PVI bit enables protected-mode virtual interrupts. • Expansion of the TSS with the software indirection bitmap. • EFLAGS.VIF bit (virtual interrupt flag). • EFLAGS.VIP bit (virtual interrupt pending flag).
2	DE—Debugging Extensions	Processor supports I/O breakpoints, including the CR4.DE bit for enabling debug extensions and optional trapping of access to the DR4 and DR5 registers.
3	PSE—Page Size Extensions	Processor supports 4-Mbyte pages, including the CR4.PSE bit for enabling page size extensions, the modified bit in page directory entries (PDEs), page directory entries, and page table entries (PTEs).
4	TSC—Time Stamp Counter	Processor supports the RDTSC (read time stamp counter) instruction, including the CR4.TSD bit that, along with the CPL, controls whether the time stamp counter can be read.
5	MSR—Model Specific Registers	Processor supports the RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions.
6	PAE—Physical Address Extension	Processor supports physical addresses greater than 32 bits, the extended page-table-entry format, an extra level in the page translation tables, and 2-MByte pages. The CR4.PAE bit enables this feature. The number of address bits is implementation specific. The Pentium® Pro processor supports 36 bits of addressing when the PAE bit is set.
7	MCE—Machine Check Exception	Processor supports the CR4.MCE bit, enabling machine check exceptions. However, this feature does not define the model-specific implementations of machine-check error logging, reporting, or processor shutdowns. Machine-check exception handlers might have to check the processor version to do model-specific processing of the exception or check for the presence of the standard machine-check feature.
8	CX8—CMPXCHG8B Instruction	Processor supports the CMPXCHG8B (compare and exchange 8 bytes) instruction.
9	APIC	Processor contains an on-chip Advanced Programmable Interrupt Controller (APIC) and it has been enabled and is available for use.
10,11 12	Reserved MTRR—Memory Type Range Registers	Processor supports machine-specific memory-type range registers (MTRRs). The MTRRs contains bit fields that indicate the processor's MTRR capabilities, including which memory types the processor supports, the number of variable MTRRs the processor supports, and whether the processor supports fixed MTRRs.
13	PGE—PTE Global Flag	Processor supports the CR4.PGE flag enabling the global bit in both PTDEs and PTEs. These bits are used to indicate translation lookaside buffer (TLB) entries that are common to different tasks and need not be flushed when control register CR3 is written.
14	MCA—Machine Check Architecture	Processor supports the MCG_CAP (machine check global capability) MSR. The MCG_CAP register indicates how many banks of error reporting MSRs the processor supports.
15	CMOV—Conditional Move and Compare Instructions	Processor supports the CMOV cc instruction and, if the FPU feature flag (bit 0) is also set, supports the FCMOV cc and FCOMI instructions.
16-22 23	Reserved MMX™ Technology	Processor supports the MMX instruction set. These instructions operate in parallel on multiple data elements (8 bytes, 4 words, or 2 doublewords) packed into quadword registers or memory locations.
24-31	Reserved	

When the input value is 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers. The encoding of these registers is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The Pentium® Pro family of processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (cleared to 0) or is reserved (set to 1).

- If a register contains valid information, the information is contained in 1 byte descriptors. Table 3-7 shows the encoding of these descriptors.

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4K-Byte Pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4M-Byte Pages, 4-way set associative, 4 entries
03H	Data TLB: 4K-Byte Pages, 4-way set associative, 64 entries
04H	Data TLB: 4M-Byte Pages, 4-way set associative, 8 entries
06H	Instruction cache: 8K Bytes, 4-way set associative, 32 byte line size
08H	Instruction cache: 16K Bytes, 4-way set associative, 32 byte line size
0AH	Data cache: 8K Bytes, 2-way set associative, 32 byte line size
0CH	Data cache: 16K Bytes, 2-way set associative, 32 byte line size
41H	Unified cache: 128K Bytes, 4-way set associative, 32 byte line size
42H	Unified cache: 256K Bytes, 4-way set associative, 32 byte line size
43H	Unified cache: 512K Bytes, 4-way set associative, 32 byte line size
44H	Unified cache: 1M Byte, 4-way set associative, 32 byte line size

The first member of the Pentium Pro processor family will return the following information about caches and TLBs when the CPUID instruction is executed with an input value of 2:

```
EAX 03 02 01 01H
EBX 0H
ECX 0H
EDX 06 04 0A 42H
```

These values are interpreted as follows:

- The least-significant byte (byte 0) of register EAX is set to 01H, indicating that the CPUID instruction needs to be executed only once with an input value of 2 to retrieve complete information about the processor's caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor contains the following:
 - 01H—A 32-entry instruction TLB (4-way set associative) for mapping 4-KByte pages.
 - 02H—A 4-entry instruction TLB (4-way set associative) for mapping 4-MByte pages.
 - 03H—A 64-entry data TLB (4-way set associative) for mapping 4-KByte pages.
- The descriptors in registers EBX and ECX are valid, but contain null descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor contains the following:
 - 42H—A 256-KByte unified cache (the L2 cache), 4-way set associative, with a 32-byte cache line size.
 - 0AH—An 8-KByte data cache (the L1 data cache), 2-way set associative, with a 32-byte cache line size.
 - 04H—An 8-entry data TLB (4-way set associative) for mapping 4M-byte pages.
 - 06H—An 8-KByte instruction cache (the L1 instruction cache), 4-way set associative, with a 32-byte cache line size.

Operands	Bytes	Clocks
	2	14 NP

Flags
None.

DAA - Decimal Adjust AL after Addition

27 DAA Decimal adjust AL after addition

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

Operands	Bytes	Clocks
	1	3 NP

Flags

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

DAS - Decimal Adjust AL after Subtraction

2F DAS Decimal adjust AL after subtraction

Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

Operands	Bytes	Clocks
	1	3 NP

Flags

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

DEC - Decrement by 1

FE /1	DEC r/m8	Decrement r/m8 by 1
FF /1	DEC r/m16	Decrement r/m16 by 1
FF /1	DEC r/m32	Decrement r/m32 by 1
48+rw	DEC r16	Decrement r16 by 1
48+rd	DEC r32	Decrement r32 by 1

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

Operands	Bytes	Clocks
r8	2	1 UV
r16	1	1 UV
r32	1	1 UV
mem	2+d(0,2)	3 UV

Flags

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

DIV - Unsigned Divide

F6 /6	DIV r/m8	Unsigned divide AX by r/m8; AL ← Quotient, AH ← Remainder
F7 /6	DIV r/m16	Unsigned divide DX:AX by r/m16; AX ← Quotient, DX ← Remainder
F7 /6	DIV r/m32	Unsigned divide EDX:EAX by r/m32 doubleword; EAX ← Quotient, EDX ← Remainder

Description

Divides (unsigned) the value in the AX register, DX:AX register pair, or EDX:EAX register pair (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size, as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/ doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

Operands	Bytes	Clocks
r8	2	17 NP
r16	2	25 NP
r32	2	41 NP
mem8	2+d(0-2)	17 NP
mem16	2+d(0-2)	25 NP
mem32	2+d(0-2)	41 NP

implied operand dividend		quotient		remainder
AX	/ byte	= AL		AH
DX:AX	/ word	= AX		DX
EDX:EAX	/ dword	= EAX		EDX

Flags

The CF, OF, SF, ZF, AF, and PF flags are undefined.

ENTER - Make Stack Frame for Procedure Parameters

C8 iw 00	ENTER imm16,0	Create a stack frame for a procedure
C8 iw 01	ENTER imm16,1	Create a nested stack frame for a procedure
C8 iw ib	ENTER imm16,imm8	Create a nested stack frame for a procedure

Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the "display area" of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits) or EBP (32 bits) register specifies the current frame pointer and whether SP (16 bits) or ESP (32 bits) specifies the stack pointer.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the EBP register onto the stack, copies the current stack pointer from the ESP register into the EBP register, and loads the ESP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See "Procedure Calls for Block-Structured Languages" in Chapter 4 of the Intel Architecture Software Developer's Manual, Volume 1, for more information about the actions of the ENTER instruction.

Operands	Bytes	Clocks
imm16, 0	3	11 NP
imm16, 1	4	15 NP
imm16, imm8	4	15+2i NP

i = imm8

Flags

None.

HLT - Halt

F4 HLT Halt

Description

Stops instruction execution and places the processor 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 a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

Operands	Bytes	Clocks
	1	4 NP

Flags

None.

IDIV - Signed Divide

F6 /7	IDIV r/m8	Signed divide AX (where AH must contain sign-extension of AL) by r/m byte. (Results: AL=Quotient, AH=Remainder)
F7 /7	IDIV r/m16	Signed divide DX:AX (where DX must contain sign-extension of AX) by r/m word. (Results: AX=Quotient, DX=Remainder)
F7 /7	IDIV r/m32	Signed divide EDX:EAX (where EDX must contain sign-extension of EAX) by r/m doubleword. (Results: EAX=Quotient, EDX=Remainder)

Description

Divides (signed) the value in the AL, AX, or EAX register by the source operand and stores the result in the AX, DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size, as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range	
Word/byte	AX	r/m8	AL	AH	-128 to +127	
Doubleword/word	DX:AX	r/m16	AX	DX	-32,768 to +32,767	
Quadword/ doubleword	EDX:EAX		r/m32	EAX	EDX	-2 ³¹ to 2 ³² - 1

Non-integral results are truncated (chopped) towards 0. The sign of the remainder is always the same as the sign of the dividend. The absolute value of the remainder is always less than the absolute value of the divisor. Overflow is indicated with the #DE (divide error) exception rather than with the OF (overflow) flag.

Operands	Bytes	Clocks
r8	2	22 NP
r16	2	30 NP
r32	2	46 NP
mem8	2+d(0-2)	22 NP
mem16	2+d(0-2)	30 NP
mem32	2+d(0-2)	46 NP

implied operand dividend	quotient	remainder
AX	/ byte	= AL AH
DX:AX	/ word	= AX DX
EDX:EAX	/ dword	= EAX EDX

Flags

The CF, OF, SF, ZF, AF, and PF flags are undefined.

IMUL - Signed Multiply

F6 /5	IMUL r/m8	AX ← AL * r/m byte
F7 /5	IMUL r/m16	DX:AX ← AX * r/m word
F7 /5	IMUL r/m32	EDX:EAX ← EAX * r/m doubleword
0F AF /r	IMUL r16,r/m16	word register ← word register * r/m word
0F AF /r	IMUL r32,r/m32	doubleword register ← doubleword register * r/m doubleword
6B /r ib	IMUL r16,r/m16,imm8	word register ← r/m16 * sign-extended immediate byte
6B /r ib	IMUL r32,r/m32,imm8	doubleword register ← r/m32 * sign-extended immediate byte
6B /r ib	IMUL r16,imm8	word register ← word register * sign-extended immediate byte
6B /r ib	IMUL r32,imm8	doubleword register ← doubleword register * sign-extended immediate byte
69 /r iw	IMUL r16,r/m16,imm16	word register ← r/m16 * immediate word
69 /r id	IMUL r32,r/m32,imm32	doubleword register ← r/m32 * immediate doubleword
69 /r iw	IMUL r16,imm16	word register ← r/m16 * immediate word
69 /r id	IMUL r32,imm32	doubleword register ← r/m32 * immediate doubleword

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

One-operand form. This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.

Two-operand form. With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.

Three-operand form. This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Accumulator Multiplies

Operands	Bytes	Clocks	
r8	2	11	NP
r16	2	11	NP
r32	2	10	NP

mem8	2+d(0-2)	11	NP
mem16	2+d(0-2)	11	NP
mem32	2+d(0-2)	10	NP

implied multiplicand		operand (multiplier)	result
AL	*	byte	= AX
AX	*	word	= DX:AX
EAX	*	dword	= EDX:EAX

2 and 3 operand Multiplies

Operands	Bytes	Clocks	
r16, imm	2+i(1,2)	10	NP
r32, imm	2+i(1,2)	10	NP
r16,r16,imm	2+i(1,2)	10	NP
r32,r32,imm	2+i(1,2)	10	NP
r16,m16,imm	2+d(0-2)+i(1,2)	10	NP
r32,m32,imm	2+d(0-2)+i(1,2)	10	NP
r16, r16	2+i(1,2)	10	NP
r32, r32	2+i(1,2)	10	NP
r16, m16	2+d(0-2)+i(1,2)	10	NP
r32, m32	2+d(0-2)+i(1,2)	10	NP

Flags

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

IN - Input from Port

E4 ib	IN AL, imm8	Input byte from imm8 I/O port address into AL
E5 ib	IN AX, imm8	Input byte from imm8 I/O port address into AX
E5 ib	IN EAX, imm8	Input byte from imm8 I/O port address into EAX
EC	IN AL, DX	Input byte from I/O port in DX into AL
ED	IN AX, DX	Input word from I/O port in DX into AX
ED	IN EAX, DX	Input doubleword from I/O port in DX into EAX

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 9, Input/Output, in the Intel Architecture Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

Operands	Bytes	Clocks
al, imm8	2	7 NP
ax, imm8	2	7 NP
eax, imm8	2	7 NP
al, dx	1	7 NP
ax, dx	1	7 NP
eax, dx	1	7 NP

Protected mode

acc, imm	2	4/21/19 NP
acc, dx	1	4/21/19 NP

cycles for: CPL <= IOPL / CPL > IOPL / V86

Flags

None.

INC - Increment by 1

FE /0	INC r/m8	Increment r/m byte by 1
FF /0	INC r/m16	Increment r/m word by 1
FF /0	INC r/m32	Increment r/m doubleword by 1
40+ rw	INC r16	Increment word register by 1
40+ rd	INC r32	Increment doubleword register by 1

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

Operands	Bytes	Clocks
r8	2	1 UV
r16	1	1 UV
r32	1	1 UV
mem	2+d(0,2)	3 UV

Flags

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

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

6C	INS m8, DX	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INS m16, DX	Input word from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INS m32, DX	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI
6C	INSB	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI
6D	INSW	Input word from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INSD	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be "DX," and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ/REPNE /REPZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 9, Input/Output, in the Intel Architecture Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

Operands	Bytes	Clocks
insb	1	9 NP
insw	1	9 NP
insd	1	9 NP

Protected Mode

1

6/24/22 NP

cycles for: $CPL \leq IOPL / CPL > IOPL / V86$

Flags

None.

INT n/INTO/INT 3 - Call to Interrupt Procedure

CC	INT 3	Interrupt 3—trap to debugger
CD ib	INT imm8	Interrupt vector number specified by immediate byte
CE	INTO	Interrupt 4—if overflow flag is 1

Description

The INT n instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 4 of the Intel Architecture Software Developer's Manual, Volume 1). The destination operand specifies an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for inter-nally generated exceptions.

The INT n instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector number 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1.

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the "normal" 2-byte opcode for INT 3 (CD03) does not have these special features. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.

The action of the INT n instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT n instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The interrupt vector number specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the interrupt vector table, and its pointers are called interrupt vectors.)

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a general protection exception (#GP); if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to three and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operands	Bytes	Clocks	
3 (const)	1	13	NP
imm8	2	16	NP
Protected mode	1	27-82 NP	

Flags

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

INVD - Invalidate Internal Caches

0F 08 INVD Flush internal caches; initiate flushing of external caches.

Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

Operands	Bytes	Clocks
	2	15 NP

Flags

None.

INVLPG - Invalidate TLB Entry

0F 01/7 INVLPG m Invalidate TLB Entry for page that contains m

Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it flushes the entire TLB. See "MOV—Move to/from Control Registers" in this chapter for further information on operations that flush the TLB.

Operands	Bytes	Clocks
mem32	5	25 NP

Flags

None.

IRET/IRETD - Interrupt Return

CF IRET Interrupt return (16-bit operand size)
CF IRETD Interrupt return (32-bit operand size)

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled "Task Linking" in Chapter 6 of the Intel Architecture Software Developer's Manual, Volume 1.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is reentered later, the code that follows the IRET instruction is executed.

Operands	Bytes	Clocks	
IRET	1	8-27	NP
IRETD	1	10-27	NP

Flags

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Jcc - Jump if Condition Is Met

77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)
72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	Jump short if CX register is 0
E3 cb	JECXZ rel8	Jump short if ECX register is 0
74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)
7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC rel16/32	Jump near if carry (CF=1)
0F 84 cw/cd	JE rel16/32	Jump near if equal (ZF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)
0F 8F cw/cd	JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE rel16/32	Jump near if greater or equal (SF=OF)
0F 8C cw/cd	JL rel16/32	Jump near if less (SF<>OF)
0F 8E cw/cd	JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 cw/cd	JNA rel16/32	Jump near if not above (CF=1 or ZF=1)
0F 82 cw/cd	JNAE rel16/32	Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB rel16/32	Jump near if not below (CF=0)
0F 87 cw/cd	JNBE rel16/32	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 cw/cd	JNC rel16/32	Jump near if not carry (CF=0)
0F 85 cw/cd	JNE rel16/32	Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG rel16/32	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE rel16/32	Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL rel16/32	Jump near if not less (SF=OF)
0F 8F cw/cd	JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 cw/cd	JNO rel16/32	Jump near if not overflow (OF=0)
0F 8B cw/cd	JNP rel16/32	Jump near if not parity (PF=0)
0F 89 cw/cd	JNS rel16/32	Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ rel16/32	Jump near if not zero (ZF=0)
0F 80 cw/cd	JO rel16/32	Jump near if overflow (OF=1)
0F 8A cw/cd	JP rel16/32	Jump near if parity (PF=1)
0F 8A cw/cd	JPE rel16/32	Jump near if parity even (PF=1)
0F 8B cw/cd	JPO rel16/32	Jump near if parity odd (PF=0)
0F 88 cw/cd	JS rel16/32	Jump near if sign (SF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the

destination operand. A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each Jcc mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The Jcc instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the Jcc instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

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

The JECXZ and JCXZ instructions differs from the other Jcc instructions because they do not check the status flags. Instead they check the contents of the ECX and CX registers, respectively, for 0. Either the CX or ECX register is chosen according to the address-size attribute. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE). They prevent entering the loop when the ECX or CX register is equal to 0, which would cause the loop to execute 2³² or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operands	Bytes	Clocks
near8	2	1 PV
near16	3	1 PV

cycles apply to jump and no jump

Flags
None.

JMP - Jump

EB cb	JMP rel8	Jump short, relative, displacement relative to next instruction
E9 cw	JMP rel16	Jump near, relative, displacement relative to next instruction
E9 cd	JMP rel32	Jump near, relative, displacement relative to next instruction
FF /4	JMP r/m16	Jump near, absolute indirect, address given in r/m16
FF /4	JMP r/m32	Jump near, absolute indirect, address given in r/m32
EA cd	JMP ptr16:16	Jump far, absolute, address given in operand
EA cp	JMP ptr16:32	Jump far, absolute, address given in operand
FF /5	JMP m16:16	Jump far, absolute indirect, address given in m16:16
FF /5 J	MP m16:32	Jump far, absolute indirect, address given in m16:32

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to -128 to $+127$ from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 6, Task Management, in the Intel Architecture Software Developer's Manual, Volume 3, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (rel8) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16 or r/m32). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (rel8, rel 16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform interprivilege level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32).

Executing a task switch with the JMP instruction, is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6, Task Management, in Intel Architecture Software Developer's Manual, Volume 3, for detailed information on the mechanics of a task switch.

Note that when you execute a task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

Operands	Bytes	Clocks
short	2	1 PV
near	3	1 PV
far	5	3 NP
r16	2	2 NP
mem16	2+d(0,2)	2 NP

mem32	2+d(4)		4	NP
r32	2	2	NP	
mem32	2+d(0,2)	2	NP	
mem48	2+d(6)		4	NP

Flags

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

JCXZ/JECXZ - Jump if CX/EAX = 0

Usage: JCXZ label
JECXZ

Description

See [JMP](#) for description and flags

Operands	Bytes	Clocks
dest	2	5/6 NP
dest	2	5/6 NP

LAHF - Load Status Flags into AH Register

9F LAHF Load: AH = EFLAGS(SF:ZF:0:AF:0:PF:1:CF)

Description

Moves the low byte of the EFLAGS register (which includes status flags SF, ZF, AF, PF, and CF) to the AH register. Reserved bits 1, 3, and 5 of the EFLAGS register are set in the AH register as shown in the "Operation" section below.

Operands	Bytes	Clocks
	1	2 NP

Flags

None (that is, the state of the flags in the EFLAGS register are not affected).

LAR - Load Access Rights Byte

0F 02 / r LAR r16,r/m16 r16 ← r/m16 masked by FF00H
0F 02 / r LAR r32,r/m32 r32 ← r/m32 masked by 00FF00H

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

When the operand size is 32 bits, the access rights for a segment descriptor include the type and DPL fields and the S, P, AVL, D/B, and G flags, all of which are located in the second double-word (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FF00H before it is loaded into the destination operand. When the operand size is 16 bits, the access rights include the type and DPL fields. Here, the two lower-order bytes of the doubleword are masked by FF00H before being loaded into the destination operand.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode.

Operands	Bytes	Clocks
r16, r16	3	8 NP
r32, r32	3	8 NP
r16, m16	3	8 NP
r32, m32	3	8 NP

Flags

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

LDS - Load Far Pointer

C5 / r	LDS r16,m16:16	Load DS: r16 with far pointer from memory
C5 / r	LDS r32,m16:32	Load DS: r32 with far pointer from memory
0F B2 / r	LSS r16,m16:16	Load SS: r16 with far pointer from memory
0F B2 / r	LSS r32,m16:32	Load SS: r32 with far pointer from memory
C4 / r	LES r16,m16:16	Load ES: r16 with far pointer from memory
C4 / r	LES r32,m16:32	Load ES: r32 with far pointer from memory
0F B4 / r	LFS r16,m16:16	Load FS: r16 with far pointer from memory
0F B4 / r	LFS r32,m16:32	Load FS: r32 with far pointer from memory
0F B5 / r	LGS r16,m16:16	Load GS: r16 with far pointer from memory
0F B5 / r	LGS r32,m16:32	Load GS: r32 with far pointer from memory

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a null selector (values 0000 through 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, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

Operands	Bytes	Clocks
reg, mem	2+d(2)	4 NP

Flags

None.

LES - Load Far Pointer

Usage: LES dest,src

Description

See [LDS](#)

Operands	Bytes	Clocks
reg, mem	2+d(2)	4 NP

LFS - Load Far Pointer

Usage: LFS dest,src

Description

See [LDS](#)

Operands	Bytes	Clocks
reg, mem	3+d(2,4)	4 NP

LGS - Load Far Pointer

Usage: LGS dest,src

Description

See [LDS](#)

Operands	Bytes	Clocks
reg, mem	3+d(2,4)	4 NP

LSS - Load Far Pointer

Usage: LSS dest,src

Description

See [LDS](#)

Operands	Bytes	Clocks
reg, mem	3+d(2,4)	4 NP

LEA - Load Effective Address

8D / r	LEA r16,m	Store effective address for m in register r16
8D / r	LEA r32,m	Store effective address for m in register r32

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

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.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

Operands	Bytes	Clocks
r16, mem	2+d(2)	1 UV
r32, mem	2+d(2)	1 UV

Flags

None.

LEAVE - High Level Procedure Exit

C9	LEAVE	Set SP to BP, then pop BP
C9	LEAVE	Set ESP to EBP, then pop EBP

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the Intel Architecture Software Developer's Manual, Volume 1, for detailed information on the use of the ENTER and LEAVE instructions.

Operands	Bytes	Clocks
	1	3 NP

Flags

None.

LGDT/LIDT - Load Global/Interrupt Descriptor Table Register

0F 01 /2	LGDT m16&32	Load m into GDTR
0F 01 /3	LIDT m16&32	Load m into IDTR

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

See "SGDT/SIDT—Store Global/Interrupt Descriptor Table Register" in this chapter for information on storing the contents of the GDTR and IDTR.

Operands	Bytes	Clocks
mem48	5	6 NP

Flags

None.

LLDT - Load Local Descriptor Table Register

0F 00 /2 LLDT r/m16

Load segment selector r/m16 into LDTR

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If the source operand is 0, the LDTR is marked invalid and all references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. Also, this instruction can only be executed in protected mode.

Operands	Bytes	Clocks
r16	3	9 NP
mem16	3+d(0-2)	9 NP

Flags

None.

LIDT - Load Interrupt Descriptor Table Register

See [LGDT/LIDT](#)

LMSW - Load Machine Status Word

0F 01 /6 LMSW r/m16 Loads r/m16 in machine status word of CR0

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on the Pentium Pro, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

Operands	Bytes	Clocks
r16	3	8 NP
mem16	3+d(0-2)	8 NP

Flags

None.

LOCK - Assert LOCK# Signal Prefix

F0 LOCK Asserts LOCK# signal for duration of the accompanying instruction

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that in later Intel Architecture processors (such as the Pentium Pro processor), locking may occur without the LOCK# signal being asserted. See Intel Architecture Compatibility below.

The LOCK prefix can be prepended only to the following instructions and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

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

Operands	Bytes	Clocks
	1	1 NP

Flags

None.

LODS/LODSB/LODSW/LODSD - Load String

AC	LODS m8	Load byte at address DS:(E)SI into AL
AD	LODS m16	Load word at address DS:(E)SI into AX
AD	LODS m32	Load doubleword at address DS:(E)SI into EAX
AC	LODSB	Load byte at address DS:(E)SI into AL
AD	LODSW	Load word at address DS:(E)SI into AX
AD	LODSD	Load doubleword at address DS:(E)SI into EAX

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:EDI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Operands	Bytes	Clocks
all variants	1	2 NP

Flags

None.

LOOP - Loop According to ECX Counter

E2 cb	LOOP rel8	Decrement count; jump short if count \neq 0
E1 cb	LOOPE rel8	Decrement count; jump short if count \neq 0 and ZF=1
E1 cb	LOOPZ rel8	Decrement count; jump short if count \neq 0 and ZF=1
E0 cb	LOOPNE rel8	Decrement count; jump short if count \neq 0 and ZF=0
E0 cb	LOOPNZ rel8	Decrement count; jump short if count \neq 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of -128 to $+127$ are allowed with this instruction.

Some forms of the loop instruction (LOOPcc) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (cc) is associated with each instruction to indicate the condition being tested for. Here, the LOOPcc instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

Operands	Bytes	Clocks
short	2	5/6 NP

Flags

None.

LOOPcc - Loop According to ECX Counter

Usage: LOOPcc label

Description

See [LOOP](#)

Operands	Bytes	Clocks
	2	7/8 NP

LSL - Load Segment Limit

0F 03 / r LSL r16,r/m16 Load: r16 ← segment limit, selector r/m16
0F 03 / r LSL r32,r/m32 Load: r32 ← segment limit, selector r/m32)

Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit "raw" limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	No
5	16-bit/32-bit task gate	No
6	16-bit interrupt gate	No
7	16-bit trap gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	No
D	Reserved	No
E	32-bit interrupt gate	No
F	32-bit trap gate	No

Operands Bytes Clocks

r16, r16	3	8	NP
r32, r32	3	8	NP
r16, m16	3+d(0,2)	8	NP
r32, m32	3+d(0,2)	8	NP

Flags

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is cleared to 0.

LTR - Load Task Register

0F 00 /3 LTR r/m16

Load r/m16 into task register

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

Operands	Bytes	Clocks	
r16	3	10	NP
mem16	3+d(0,2)	10	NP

Flags

None.

MOV - Move

88 / r	MOV r/m8,r8	Move r8 to r/m8
89 / r	MOV r/m16,r16	Move r16 to r/m16
89 / r	MOV r/m32,r32	Move r32 to r/m32
8A / r	MOV r8,r/m8	Move r/m8 to r8
8B / r	MOV r16,r/m16	Move r/m16 to r16
8B / r	MOV r32,r/m32	Move r/m32 to r32
8C / r	MOV r/m16,Sreg**	Move segment register to r/m16
8E / r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL, moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX, moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX, moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ rb	MOV r8,imm8	Move imm8 to r8
B8+ rw	MOV r16,imm16	Move imm16 to r16
B8+ rd	MOV r32,imm32	Move imm32 to r32
C6 / 0	MOV r/m8,imm8	Move imm8 to r/m8
C7 / 0	MOV r/m16,imm16	Move imm16 to r/m16
C7 / 0	MOV r/m32,imm32	Move imm32 to r/m32

NOTES:

*The moffs8, moffs16, and moffs32 operands specify a simple offset relative to the segment base, where 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following "Description" section for further information).

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, or a doubleword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A null segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, stack-pointer value) before an interrupt occurs 1. The LSS instruction offers a more efficient method of loading the SS and ESP registers.

When operating in 32-bit mode and moving data between a segment register and a general-purpose register, the Intel Architecture 32-bit processors do not require the use of the 16-bit operand-size prefix (a byte with the value 66H) with this instruction, but most assemblers will insert it if the standard form of the instruction is used (for example, MOV DS, AX). The processor will execute this instruction correctly, but it will usually require an extra clock. With most assemblers, using the instruction form MOV DS, EAX will avoid this unneeded 66H prefix. When the processor executes the instruction with a 32-bit general-purpose register, it assumes that the 16 least-significant bits of the general-purpose register are the destination or source operand. If the register is a destination operand, the resulting value in the two high-

order bytes of the register is implementation dependent. For the Pentium Pro processor, the two high-order bytes are filled with zeros; for earlier 32-bit Intel Architecture processors, the two high order bytes are undefined.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0-2)	1 UV
reg, mem	2+d(0-2)	1 UV
mem, imm	2+d(0-2)+i(1,2)	1 UV*
reg, imm	2+i(1,2)	1 UV
acc, mem	3	1 UV
mem, acc	3	1 UV

* = not pairable if there is a displacement and immediate

Segment Register Moves

Real Mode

seg, r16	2	2-11	NP
seg, m16	2+d(0,2)	3-12	NP
r16, seg	2	1	NP
m16, seg	2+d(0,2)	1	NP

Protected Mode Differences

seg, r16	2	2-11*	NP
seg, m16	2+d(0,2)	3-12*	NP

* = add 8 if new descriptor; add 6 if SS

MOVE to/from special registers

r32, cr32	3	4	NP
cr32, r32	3	12/22*	NP
r32, dr32	3	2/12*	NP
dr32, r32	3	11/12*	NP
r32, tr32	3	-	NP
tr32, r32	3	-	NP

* = cycles depend on which special register

Flags

None.

MOV - Move to/from Control Registers

0F 22 / r	MOV CR0, r32	Move r32 to CR0
0F 22 / r	MOV CR2, r32	Move r32 to CR2
0F 22 / r	MOV CR3, r32	Move r32 to CR3
0F 22 / r	MOV CR4, r32	Move r32 to CR4
0F 20 / r	MOV r32, CR0	Move CR0 to r32
0F 20 / r	MOV r32, CR2	Move CR2 to r32
0F 20 / r	MOV r32, CR3	Move CR3 to r32
0F 20 / r	MOV r32, CR4	Move CR4 to r32

Description

Moves the contents of a control register (CR0, CR2, CR3, or CR4) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See "Control Registers" in Chapter 2 of the Intel Architecture Software Developer's Manual, Volume 3, for a detailed description of the flags and fields in the control registers.)

When loading a control register, a program should not attempt to change any of the reserved bits; that is, always set reserved bits to the value previously read.

At the opcode level, the reg field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the mod field are always 11B. The r/m field specifies the general-purpose register loaded or read.

These instructions have the following side effects:

- When writing to control register CR3, all non-global TLB entries are flushed (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the Intel Architecture Software Developer's Manual, Volume 3).

Operands	Bytes	Clocks
See MOV		

Flags

The OF, SF, ZF, AF, PF, and CF flags are undefined.

MOV - Move to/from Debug Registers

0F 21 / r	MOV r32, DR0-DR7	Move debug register to r32
0F 23 / r	MOV DR0-DR7, r32	Move r32 to debug register

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See Chapter 14, Debugging and Performance Monitoring, of the Intel Architecture Software Developer's Manual, Volume 3, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE set in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the Intel Architecture beginning with the Pentium processor.)

At the opcode level, the reg field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the mod field are always 11. The r/m field specifies the general-purpose register loaded or read.

Operands	Bytes	Clocks
See MOV		

Flags

The OF, SF, ZF, AF, PF, and CF flags are undefined.

MOVS/MOVSb/MOVSsw/MOVSd - Move Data from String to String

A4	MOVS m8, m8	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVS m16, m16	Move word at address DS:(E)SI to address ES:(E)DI
A5	MOVS m32, m32	Move doubleword at address DS:(E)SI to address ES:(E)DI
A4	MOVSb	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVSsw	Move word at address DS:(E)SI to address ES:(E)DI
A5	MOVSd	Move doubleword at address DS:(E)SI to address ES:(E)DI

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be over-ridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct location. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSb (byte move), MOVSsw (word move), or MOVSd (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incremented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The MOVS, MOVSb, MOVSsw, and MOVSd instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE /REPZ—Repeat String Operation Prefix" in this chapter) for block moves of ECX bytes, words, or doublewords.

Variations	Bytes	Clocks
movsb	1	4 NP
movsw	1	4 NP
movsd	1	4 NP
rep movsb	2	3+n NP
rep movsw	2	3+n NP
rep movsd	2	3+n NP

(n = count of bytes, words or dwords)

Flags

None.

MOVSX - Move with Sign-Extension

0F BE / r	MOVSX r16,r/m8	Move byte to word with sign-extension
0F BE / r	MOVSX r32,r/m8	Move byte to doubleword, sign-extension
0F BF / r	MOVSX r32,r/m16	Move word to doubleword, sign-extension

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 6-5 in the Intel Architecture Software Developer's Manual, Volume 1). The size of the converted value depends on the operand-size attribute.

Operands	Bytes	Clocks
----------	-------	--------

reg, reg	3	3 NP
----------	---	------

reg, mem	3+d(0,1,2,4)	3 NP
----------	--------------	------

(Note: destination reg is 16 or 32-bits; source is 8 or 16 bits)

Flags

None.

MOVZX - Move with Zero-Extend

0F B6 / r	MOVZX r16,r/m8	Move byte to word with zero-extension
0F B6 / r	MOVZX r32,r/m8	Move byte to doubleword, zero-extension
0F B7 / r	MOVZX r32,r/m16	Move word to doubleword, zero-extension

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Operands	Bytes	Clocks
reg, reg	3	3 NP
reg, mem	3+d(0,1,2,4)	3 NP

(Note: destination reg is 16 or 32-bits; source is 8 or 16 bits)

Flags

None.

MUL - Unsigned Multiply

F6 /4	MUL r/m8	Unsigned multiply ($AX \leftarrow AL * r/m8$)
F7 /4	MUL r/m16	Unsigned multiply ($DX:AX \leftarrow AX * r/m16$)
F7 /4	MUL r/m32	Unsigned multiply ($EDX:EAX \leftarrow EAX * r/m32$)

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in the following table.

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

Operands	Bytes	Clocks
r8	2	11 NP
r16	2	11 NP
r32	2	10 NP
mem8	2+d(0-2)	11 NP
mem16	2+d(0-2)	11 NP
mem32	2+d(0-2)	10 NP

implied multiplicand	operand (multiplier)	result
AL *	byte	= AX
AX *	word	= DX:AX
EAX *	dword	= EDX:EAX

Flags

The OF and CF flags are cleared to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

NEG - Two's Complement Negation

F6 /3	NEG r/m8	Two's complement negate r/m8
F7 /3	NEG r/m16	Two's complement negate r/m16
F7 /3	NEG r/m32	Two's complement negate r/m32

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

Operands	Bytes	Clocks
reg	2	1 NP
mem	2+d(0-2)	3 NP

Flags

The CF flag cleared to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

NOP - No Operation

90 NOP No operation

Description

Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

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

Operands	Bytes	Clocks
	1	1 UV

Flags

None.

NOT - One's Complement Negation

F6 /2	NOT r/m8	Reverse each bit of r/m8
F7 /2	NOT r/m16	Reverse each bit of r/m16
F7 /2	NOT r/m32	Reverse each bit of r/m32

Description

Performs a bitwise NOT operation (each 1 is cleared to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

Operands	Bytes	Clocks
reg	2	1 NP
mem	2+d(0-2)	3 NP

Flags

None.

OR - Logical Inclusive OR

0C ib	OR AL, imm8 AL	OR imm8
0D iw	OR AX, imm16 AX	OR imm16
0D id	OR EAX, imm32 EAX	OR imm32
80 /1 ib	OR r/m8, imm8 r/m8	OR imm8
81 /1 iw	OR r/m16, imm16 r/m16	OR imm16
81 /1 id	OR r/m32, imm32 r/m32	OR imm32
83 /1 ib	OR r/m16, imm8 r/m16	OR imm8 (sign-extended)
83 /1 id	OR r/m32, imm8 r/m32	OR imm8 (sign-extended)
08 / r	OR r/m8, r8 r/m8	OR r8
09 / r	OR r/m16, r16 r/m16	OR r16
09 / r	OR r/m32, r32 r/m32	OR r32
0A / r	OR r8, r/m8 r8	OR r/m8
0B / r	OR r16, r/m16 r16	OR r/m16
0B / r	OR r32, r/m32 r32	OR r/m32

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0,2)	3 UV
reg, mem	2+d(0,2)	2 UV
reg, imm	2+i(1,2)	1 UV
mem, imm	2+d(0,2)+i(1,2)	3 UV*
acc, imm	1+i(1,2)	1 UV

* = not pairable if there is a displacement and immediate

Flags

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

OUT - Output to Port

E6 ib	OUT imm8, AL	Output byte in AL to I/O port address imm8
E7 ib	OUT imm8, AX	Output word in AX to I/O port address imm8
E7 ib	OUT imm8, EAX	Output doubleword in EAX to I/O port address imm8
EE	OUT DX, AL	Output byte in AL to I/O port address in DX
EF	OUT DX, AX	Output word in AX to I/O port address in DX
EF	OUT DX, EAX	Output doubleword in EAX to I/O port address in DX

Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 9, Input/Output, in the Intel Architecture Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

Operands	Bytes	Clocks
imm8, al	2	12 NP
imm8, ax	2	12 NP
imm8, eax	2	12 NP
dx, al	1	12 NP
dx, ax	1	12 NP
dx, eax	1	12 NP

Protected Mode

imm8, acc	2	9/26/24 NP
dx, acc	1	9/26/24 NP

cycles for: CPL <= IOPL / CPL > IOPL / V86

Flags

None.

OUTS/OUTSB/OUTSW/OUTSD - Output String to Port

6E	OUTS DX, m8	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTS DX, m16	Output word from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTS DX, m32	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX
6E	OUTSB	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTSW	Output word from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTSD	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:EDI or the DS:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the (E)SI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ/REPNE /REPZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 9, Input/Output, in the Intel Architecture Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

Variations	Bytes	Clocks
outsb	1	13 NP
outsw	1	13 NP
outsd	1	13 NP

Protected Mode

1

10/27/25 NP

cycles for: $CPL \leq IOPL / CPL > IOPL / V86$

Flags

None.

POP - Pop a Value from the Stack

8F /0	POP m16	Pop top of stack into m16; increment stack pointer
8F /0	POP m32	Pop top of stack into m32; increment stack pointer
58+ rw	POP r16	Pop top of stack into r16; increment stack pointer
58+ rd	POP r32	Pop top of stack into r32; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer

Description

Loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits—the source address size), and the operand-size attribute of the current code segment determines the amount the stack pointer is incremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is incremented by 4 and, if they are 16, the 16-bit SP register is incremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the destination operand.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is null.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt 1. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

Operands	Bytes	Clocks
reg	1	1 UV
mem	2+d(0-2)	3 NP
seg	1	3 NP
FS/GS	2	3 NP

Protected Mode

CS/DS/ES	1	3-12	NP
SS	1	8-17	NP
FS/GS	2	3-12	NP

Flags

None.

POPA/POPAD - Pop All General-Purpose Registers

61	POPA	Pop DI, SI, BP, BX, DX, CX, and AX
61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX

Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

Variations	Bytes	Clocks
popa	1	5 NP
popad	1	5 NP

popa =	pop di, si, bp, sp, bx, dx, cx, ax
popad =	pop edi, esi, ebp, esp, ebx, edx, ecx, eax (sp and esp are discarded)

Flags

None.

POPF/POPFD - Pop Stack into EFLAGS Register

9D POPF Pop top of stack into lower 16 bits of EFLAGS
9D POPFD Pop top of stack into EFLAGS

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). (These instructions reverse the operation of the PUSHF/PUSHFD instructions.)

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16 and the POPFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPF is used and to 32 when POPFD is used. Others may treat these mnemonics as synonyms (POPF/POPFD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

The effect of the POPF/POPFD instructions on the EFLAGS register changes slightly, depending on the mode of operation of the processor. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, which is equivalent to privilege level 0), all the non-reserved flags in the EFLAGS register except the VIP, VIF, and VM flags can be modified. The VIP and VIF flags are cleared, and the VM flag is unaffected.

When operating in protected mode, with a privilege level greater than 0, but less than or equal to IOPL, all the flags can be modified except the IOPL field and the VIP, VIF, and VM flags. Here, the IOPL flags are unaffected, the VIP and VIF flags are cleared, and the VM flag is unaffected. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

When operating in virtual-8086 mode, the I/O privilege level (IOPL) must be equal to 3 to use POPF/POPFD instructions and the VM, RF, IOPL, VIP, and VIF flags are unaffected. If the IOPL is less than 3, the POPF/POPFD instructions cause a general-protection exception (#GP).

See the section titled "EFLAGS Register" in Chapter 3 of the Intel Architecture Software Developer's Manual, Volume 1, for information about the EFLAGS registers.

Variations	Bytes	Clocks
popf	1	6 NP
popfd	1	6 NP

Protected Mode		
popf	1	4 NP
popfd	1	4 NP

Flags

All flags except the reserved bits and the VM bit.

PUSH - Push Word or Doubleword Onto the Stack

FF /6	PUSH r/m16	Push r/m16
FF /6	PUSH r/m32	Push r/m32
50+ rw	PUSH r16	Push r16
50+ rd	PUSH r32	Push r32
6A	PUSH imm8	Push imm8
68	PUSH imm16	Push imm16
68	PUSH imm32	Push imm32
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
0F A8	PUSH GS	Push GS

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits), and the operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4 and, if they are 16, the 16-bit SP register is decremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the source operand.) Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer is not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operands	Bytes	Clocks	
reg	1	1	UV
mem	2+d(0-2)	2	NP
seg	1	1	NP
imm	1+i(1,2)	1	NP
FS/GS	2	1	NP

Flags

None.

PUSHA/PUSHAD - Push All General-Purpose Registers

60	PUSHA	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). (These instructions perform the reverse operation of the POPA/POPAD instructions.) The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Variations	Bytes	Clocks
pusha	1	5 NP
pushad	1	5 NP

pusha = push ax, cx, dx, bx, sp, bp, si, di,
pushad = push eax, ecx, edx, ebx, esp, ebp, esi, edi

Flags

None.

PUSHF/PUSHFD - Push EFLAGS Register onto the Stack

9C	PUSHF	Push lower 16 bits of EFLAGS
9C	PUSHFD	Push EFLAGS

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. (These instructions reverse the operation of the POPF/POPFD instructions.) When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See the section titled "EFLAGS Register" in Chapter 3 of the Intel Architecture Software Developer's Manual, Volume 1, for information about the EFLAGS registers.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

When in virtual-8086 mode and the I/O privilege level (IOPL) is less than 3, the PUSHF/PUSHFD instruction causes a general protection exception (#GP).

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Variations	Bytes	Clocks
pushf	1	9 NP
pushfd	1	9 NP

Protected Mode		
pushf	1	3 NP
pushfd	1	3 NP

Flags

None.

RCL - Rotate Bits Left with CF

D0 /2	RCL r/m8,1	Rotate 9 bits (CF, r/m8) left once
D2 /2	RCL r/m8,CL	Rotate 9 bits (CF, r/m8) left CL times
C0 /2 ib	RCL r/m8,imm8	Rotate 9 bits (CF, r/m8) left imm8 times
D1 /2	RCL r/m16,1	Rotate 17 bits (CF, r/m16) left once
D3 /2	RCL r/m16,CL	Rotate 17 bits (CF, r/m16) left CL times
C1 /2 ib	RCL r/m16,imm8	Rotate 17 bits (CF, r/m16) left imm8 times
D1 /2	RCL r/m32,1	Rotate 33 bits (CF, r/m32) left once
D3 /2	RCL r/m32,CL	Rotate 33 bits (CF, r/m32) left CL times
C1 /2 ib	RCL r/m32,imm8	Rotate 33 bits (CF, r/m32) left imm8 times
D0 /3	RCR r/m8,1	Rotate 9 bits (CF, r/m8) right once
D2 /3	RCR r/m8,CL	Rotate 9 bits (CF, r/m8) right CL times
C0 /3 ib	RCR r/m8,imm8	Rotate 9 bits (CF, r/m8) right imm8 times
D1 /3	RCR r/m16,1	Rotate 17 bits (CF, r/m16) right once
D3 /3	RCR r/m16,CL	Rotate 17 bits (CF, r/m16) right CL times
C1 /3 ib	RCR r/m16,imm8	Rotate 17 bits (CF, r/m16) right imm8 times
D1 /3	RCR r/m32,1	Rotate 33 bits (CF, r/m32) right once
D3 /3	RCR r/m32,CL	Rotate 33 bits (CF, r/m32) right CL times
C1 /3 ib	RCR r/m32,imm8	Rotate 33 bits (CF, r/m32) right imm8 times
D0 /0	ROL r/m8,1	Rotate 8 bits r/m8 left once
D2 /0	ROL r/m8,CL	Rotate 8 bits r/m8 left CL times
C0 /0 ib	ROL r/m8,imm8	Rotate 8 bits r/m8 left imm8 times
D1 /0	ROL r/m16,1	Rotate 16 bits r/m16 left once
D3 /0	ROL r/m16,CL	Rotate 16 bits r/m16 left CL times
C1 /0 ib	ROL r/m16,imm8	Rotate 16 bits r/m16 left imm8 times
D1 /0	ROL r/m32,1	Rotate 32 bits r/m32 left once
D3 /0	ROL r/m32,CL	Rotate 32 bits r/m32 left CL times
C1 /0 ib	ROL r/m32,imm8	Rotate 32 bits r/m32 left imm8 times
D0 /1	ROR r/m8,1	Rotate 8 bits r/m8 right once
D2 /1	ROR r/m8,CL	Rotate 8 bits r/m8 right CL times
C0 /1 ib	ROR r/m8,imm8	Rotate 8 bits r/m16 right imm8 times
D1 /1	ROR r/m16,1	Rotate 16 bits r/m16 right once
D3 /1	ROR r/m16,CL	Rotate 16 bits r/m16 right CL times
C1 /1 ib	ROR r/m16,imm8	Rotate 16 bits r/m16 right imm8 times
D1 /1	ROR r/m32,1	Rotate 32 bits r/m32 right once
D3 /1	ROR r/m32,CL	Rotate 32 bits r/m32 right CL times
C1 /1 ib	ROR r/m32,imm8	Rotate 32 bits r/m32 right imm8 times

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location (see Figure 6-10 in the Intel Architecture Software Developer's Manual, Volume 1). The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location (see Figure 6-10 in the Intel Architecture Software Developer's Manual, Volume 1).

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag (see Figure 6-10 in the Intel Architecture Software Developer's Manual, Volume 1). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see Figure 6-10 in the Intel Architecture Software Developer's Manual, Volume 1). For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the

exclusive OR of the two most-significant bits of the result.

Operands	Bytes	Clocks
reg, 1	2	1 PU
mem, 1	2+d(0,2)	3 PU
reg, cl	2	7-24 NP
mem, cl	2+d(0,2)	9-26 NP
reg, imm	3	8-25 NP
mem, imm	3+d(0,2)	10-27 NP

Flags

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see "Description" above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

RCR - Rotate Bits Right with CF

Usage: RCR dest, count

Description

See [RCL](#)

Operands	Bytes	Clocks
reg, 1	2	1 PU
mem, 1	2+d(0,2)	3 PU
reg, cl	2	7-24 NP
mem, cl	2+d(0,2)	9-26 NP
reg, imm	3	8-25 NP
mem, imm	3+d(0,2)	10-27 NP

ROL - Rotate Bits Left

Usage: ROL dest, count

Description

See [RCL](#)

Operands	Bytes	Clocks
reg, 1	2	1 PU
mem, 1	2+d(0,2)	3 PU
reg, cl	2	4 NP
mem, cl	2+d(0,2)	4 NP
reg, imm	3	1 PU
mem, imm	3+d(0,2)	3 PU*

* = not pairable if there is a displacement and immediate

ROR - Rotate Bits Right

Usage: ROR dest, count

Description

See [RCL](#)

Operands	Bytes	Clocks
reg, 1	2	1 PU
mem, 1	2+d(0,2)	3 PU
reg, cl	2	4 NP
mem, cl	2+d(0,2)	4 NP
reg, imm	3	1 PU
mem, imm	3+d(0,2)	3 PU*

* = not pairable if there is a displacement and immediate

RDMSR - Read from Model Specific Register

0F 32 RDMSR Load MSR specified by ECX into EDX:EAX

Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. If less than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, Model-Specific Registers (MSRs), in the Intel Architecture Software Developer's Manual, Volume 3, lists all the MSRs that can be read with this instruction and their addresses.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

Operands	Bytes	Clocks
	2	20-24 NP

Flags

None.

RDTSC - Read Time-Stamp Counter

0F 31 RDTSC Read time-stamp counter into EDX:EAX

Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the Intel Architecture in the Pentium processor.

Operands	Bytes	Clocks
	2	6-7 NP

Flags

None.

REP/REPE/REPZ/REPNE/REPNZ - Repeat String Operation Prefix

F3 6C	REP INS r/m8, DX	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS r/m16,DX	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS r/m32,DX	Input (E)CX doublewords from port DX into ES:[(E)DI]
F3 A4	REP MOVS m8,m8	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS m16,m16	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS m32,m32	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX, r/m8	Output (E)CX bytes from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX, r/m16	Output (E)CX words from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX, r/m32	Output (E)CX doublewords from DS:[(E)SI] to port DX
F3 AC	REP LODS AL	Load (E)CX bytes from DS:[(E)SI] to AL
F3 AD	REP LODS AX	Load (E)CX words from DS:[(E)SI] to AX
F3 AD	REP LODS EAX	Load (E)CX doublewords from DS:[(E)SI] to EAX
F3 AA	REP STOS m8	Fill (E)CX bytes at ES:[(E)DI] with AL
F3 AB	REP STOS m16	Fill (E)CX words at ES:[(E)DI] with AX
F3 AB	REP STOS m32	Fill (E)CX doublewords at ES:[(E)DI] with EAX
F3 A6	REPE CMPS m8,m8	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS m16,m16	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI]
F3 A7	REPE CMPS m32,m32	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI]
F3 AE	REPE SCAS m8	Find non-AL byte starting at ES:[(E)DI]
F3 AF	REPE SCAS m16	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS m32	Find non-EAX doubleword starting at ES:[(E)DI]
F2 A6	REPNE CMPS m8,m8	Find matching bytes in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS m16,m16	Find matching words in ES:[(E)DI] and DS:[(E)SI]
F2 A7	REPNE CMPS m32,m32	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI]
F2 AE	REPNE SCAS m8	Find AL, starting at ES:[(E)DI]
F2 AF	REPNE SCAS m16	Find AX, starting at ES:[(E)DI]
F2 AF	REPNE SCAS m32	Find EAX, starting at ES:[(E)DI]

Description

Repeats a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

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.

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see the following table). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated

on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

Operands	Bytes	Clocks
-----------------	--------------	---------------

See MOVS	(rep movs)	move block
--------------------------	------------	------------

See STOS	(rep stos)	fill block
--------------------------	------------	------------

See CMPS	(repe cmps)	find non-matching memory items
--------------------------	-------------	--------------------------------

See CMPS	(repe scas)	find non-acc matching byte in memory
--------------------------	-------------	--------------------------------------

Flags

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

RET - Return from Procedure

C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 iw	RET imm16	Near return to calling procedure and pop imm16 bytes from stack
CA iw	RET imm16	Far return to calling procedure and pop imm16 bytes from stack

Description

Transfers program control to a return address located on the top of 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 instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 4 of the Intel Architecture Software Developer's Manual, Volume 1, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

Varations	Bytes	Clocks
retn	1	2 NP
retn imm16	1+d(2)	3 NP
retf	1	4 NP

retf imm16 1+d(2) 4 NP

Flags

None.

RSM - Resume from System Management Mode

0F AA RSM Resume operation of interrupted program

Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SSM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium ® and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

See Chapter 11, System Management Mode (SMM), in the Intel Architecture Software Developer's Manual, Volume 3, for more information about SMM and the behavior of the RSM instruction.

Operands	Bytes	Clocks
	2	83 NP

Flags

All.

SAHF - Store AH into Flags

9E SAHF Loads SF, ZF, AF, PF, and CF from AH into
EFLAGS register

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the "Operation" section below.

Operands	Bytes	Clocks
	1	2 NP

Flags

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

SAL - Shift Arithmetic Left

D0 /4	SAL r/m8,1	Multiply r/m8 by 2, once
D2 /4	SAL r/m8,CL	Multiply r/m8 by 2, CL times
C0 /4 ib	SAL r/m8,imm8	Multiply r/m8 by 2, imm8 times
D1 /4	SAL r/m16,1	Multiply r/m16 by 2, once
D3 /4	SAL r/m16,CL	Multiply r/m16 by 2, CL times
C1 /4 ib	SAL r/m16,imm8	Multiply r/m16 by 2, imm8 times
D1 /4	SAL r/m32,1	Multiply r/m32 by 2, once
D3 /4	SAL r/m32,CL	Multiply r/m32 by 2, CL times
C1 /4 ib	SAL r/m32,imm8	Multiply r/m32 by 2, imm8 times
D0 /7	SAR r/m8,1	Signed divide* r/m8 by 2, once
D2 /7	SAR r/m8,CL	Signed divide* r/m8 by 2, CL times
C0 /7 ib	SAR r/m8,imm8	Signed divide* r/m8 by 2, imm8 times
D1 /7	SAR r/m16,1	Signed divide* r/m16 by 2, once
D3 /7	SAR r/m16,CL	Signed divide* r/m16 by 2, CL times
C1 /7 ib	SAR r/m16,imm8	Signed divide* r/m16 by 2, imm8 times
D1 /7	SAR r/m32,1	Signed divide* r/m32 by 2, once
D3 /7	SAR r/m32,CL	Signed divide* r/m32 by 2, CL times
C1 /7 ib	SAR r/m32,imm8	Signed divide* r/m32 by 2, imm8 times
D0 /4	SHL r/m8,1	Multiply r/m8 by 2, once
D2 /4	SHL r/m8,CL	Multiply r/m8 by 2, CL times
C0 /4 ib	SHL r/m8,imm8	Multiply r/m8 by 2, imm8 times
D1 /4	SHL r/m16,1	Multiply r/m16 by 2, once
D3 /4	SHL r/m16,CL	Multiply r/m16 by 2, CL times
C1 /4 ib	SHL r/m16,imm8	Multiply r/m16 by 2, imm8 times
D1 /4	SHL r/m32,1	Multiply r/m32 by 2, once
D3 /4	SHL r/m32,CL	Multiply r/m32 by 2, CL times
C1 /4 ib	SHL r/m32,imm8	Multiply r/m32 by 2, imm8 times
D0 /5	SHR r/m8,1	Unsigned divide r/m8 by 2, once
D2 /5	SHR r/m8,CL	Unsigned divide r/m8 by 2, CL times
C0 /5 ib	SHR r/m8,imm8	Unsigned divide r/m8 by 2, imm8 times
D1 /5	SHR r/m16,1	Unsigned divide r/m16 by 2, once
D3 /5	SHR r/m16,CL	Unsigned divide r/m16 by 2, CL times
C1 /5 ib	SHR r/m16,imm8	Unsigned divide r/m16 by 2, imm8 times
D1 /5	SHR r/m32,1	Unsigned divide r/m32 by 2, once
D3 /5	SHR r/m32,CL	Unsigned divide r/m32 by 2, CL times
C1 /5 ib	SHR r/m32,imm8	Unsigned divide r/m32 by 2, imm8 times

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 6-6 in the Intel Architecture Software Developer's Manual, Volume 1).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 6-7 in the Intel Architecture Software Developer's Manual, Volume 1); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 6-8 in the Intel Architecture Software Developer's Manual, Volume 1).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the

destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is cleared to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

Operands	Bytes	Clocks
reg, 1	2	1 PU
mem, 1	2+d(0,2)	3 PU
reg, cl	2	4 NP
mem, cl	2+d(0,2)	4 NP
reg, imm	3	1 PU
mem, imm	3+d(0,2)	3 PU*

* = not pairable if there is a displacement and immediate

sal = shift arithmetic left	sar = shift arithmetic right
shl = shift left (same as sal)	shr = shift right

Flags

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

SAR - Shift Arithmetic Right

See [SAL](#)

SHL - Shift Left

See [SAL](#)

SHR - Shift Right

See [SAL](#)

SBB - Integer Subtraction with Borrow

1C ib	SBB AL, imm8	Subtract with borrow imm8 from AL
1D iw	SBB AX, imm16	Subtract with borrow imm16 from AX
1D id	SBB EAX, imm32	Subtract with borrow imm32 from EAX
80 /3 ib	SBB r/m8, imm8	Subtract with borrow imm8 from r/m8
81 /3 iw	SBB r/m16, imm16	Subtract with borrow imm16 from r/m16
81 /3 id	SBB r/m32, imm32	Subtract with borrow imm32 from r/m32
83 /3 ib	SBB r/m16, imm8	Subtract with borrow sign-extended imm8 from r/m16
83 /3 id	SBB r/m32, imm8	Subtract with borrow sign-extended imm8 from r/m32
18 / r	SBB r/m8, r8	Subtract with borrow r8 from r/m8
19 / r	SBB r/m16, r16	Subtract with borrow r16 from r/m16
19 / r	SBB r/m32, r32	Subtract with borrow r32 from r/m32
1A / r	SBB r8, r/m8	Subtract with borrow r/m8 from r8
1B / r	SBB r16, r/m16	Subtract with borrow r/m16 from r16
1B / r	SBB r32, r/m32	Subtract with borrow r/m32 from r32

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

Operands	Bytes	Clocks
reg, reg	2	1 PU
mem, reg	2+d(0,2)	3 PU
reg, mem	2+d(0,2)	2 PU
reg, imm	2+i(1,2)	1 PU
mem, imm	2+d(0,2)+i(1,2)	3 PU*
acc, imm	1+i(1,2)	1 PU

* = not pairable if there is a displacement and immediate

Flags

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

SCAS/SCASB/SCASW/SCASD - Scan String

AE	SCAS m8	Compare AL with byte at ES:(E)DI and set status flags
AF	SCAS m16	Compare AX with word at ES:(E)DI and set status flags
AF	SCAS m32	Compare EAX with doubleword at ES(E)DI and set status flags
AE	SCASB	Compare AL with byte at ES:(E)DI and set status flags
AF	SCASW	Compare AX with word at ES:(E)DI and set status flags
AF	SCASD	Compare EAX with doubleword at ES:(E)DI and set status flags

Description

Compares the byte, word, or double word specified with the memory operand with the value in the AL, AX, or EAX register, and sets the status flags in the EFLAGS register according to the results. The memory operand address is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operand form (specified with the SCAS mnemonic) allows the memory operand to be specified explicitly. Here, the memory operand should be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (the AL register for byte comparisons, AX for word comparisons, and EAX for doubleword comparisons). This explicit-operand form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the SCAS instructions. Here also ES:(E)DI is assumed to be the memory operand and the AL, AX, or EAX register is assumed to be the register operand. The size of the two operands is selected with the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for double-word operations.

The SCAS, SCASB, SCASW, and SCASD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Variations	Bytes	Clocks
scasb	1	4 NP
scasw	1	4 NP
scasd	1	4 NP
repX scasb	2	8+4n NP
repX scasw	2	8+4n NP
repX scasd	2	8+4n NP

repX = repe or repz or repne or repnz
(n = count of bytes, words or dwords)

Flags

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

SETcc - Set Byte on Condition

0F 97	SETA r/m8	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE r/m8	Set byte if above or equal (CF=0)
0F 92	SETB r/m8	Set byte if below (CF=1)
0F 96	SETBE r/m8	Set byte if below or equal (CF=1 or ZF=1)
0F 92	SETC r/m8	Set if carry (CF=1)
0F 94	SETE r/m8	Set byte if equal (ZF=1)
0F 9F	SETG r/m8	Set byte if greater (ZF=0 and SF=OF)
0F 9D	SETGE r/m8	Set byte if greater or equal (SF=OF)
0F 9C	SETL r/m8	Set byte if less (SF<>OF)
0F 9E	SETLE r/m8	Set byte if less or equal (ZF=1 or SF<>OF)
0F 96	SETNA r/m8	Set byte if not above (CF=1 or ZF=1)
0F 92	SETNAE r/m8	Set byte if not above or equal (CF=1)
0F 93	SETNB r/m8	Set byte if not below (CF=0)
0F 97	SETNBE r/m8	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC r/m8	Set byte if not carry (CF=0)
0F 95	SETNE r/m8	Set byte if not equal (ZF=0)
0F 9E	SETNG r/m8	Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE r/m8	Set if not greater or equal (SF<>OF)
0F 9D	SETNL r/m8	Set byte if not less (SF=OF)
0F 9F	SETNLE r/m8	Set byte if not less or equal (ZF=0 and SF=OF)
0F 91	SETNO r/m8	Set byte if not overflow (OF=0)
0F 9B	SETNP r/m8	Set byte if not parity (PF=0)
0F 99	SETNS r/m8	Set byte if not sign (SF=0)
0F 95	SETNZ r/m8	Set byte if not zero (ZF=0)
0F 90	SETO r/m8	Set byte if overflow (OF=1)
0F 9A	SETP r/m8	Set byte if parity (PF=1)
0F 9A	SETPE r/m8	Set byte if parity even (PF=1)
0F 9B	SETPO r/m8	Set byte if parity odd (PF=0)
0F 98	SETS r/m8	Set byte if sign (SF=1)
0F 94	SETZ r/m8	Set byte if zero (ZF=1)

Description

Set the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (cc) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SETcc instruction opcodes have alternate mnemonics. For example, the SETG (set byte if greater) and SETNLE (set if not less or equal) both have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, EFLAGS Condition Codes, in the Intel Architecture Software Developer's Manual, Volume 1, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SETcc instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

Operands	Bytes	Clocks
r8	3	1/2 NP
mem8	3+d(0-2)	1/2 NP

Cycles are for: true/false

setCC = one of:

seta	setae	setb	setbe	setc	sete
setg	setge	setl	setle	setna	setnae
setnb	setnbe	setnc	setne	setng	setnge
setnl	setnle	setno	setnp	setns	setnz

seto setp setpe setpo sets setz

Flags

None.

SGDT/SIDT - Store Global/Interrupt Descriptor Table Register

0F 01 /0	SGDT m	Store GDTR to m
0F 01 /1	SIDT m	Store IDTR to m

Description

Stores the contents of the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the lower 2 bytes of the memory location and the 32-bit base address is stored in the upper 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the lower 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

The SGDT and SIDT instructions are only useful in operating-system software; however, they can be used in application programs without causing an exception to be generated.

See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in this chapter for information on loading the GDTR and IDTR.

Operands	Bytes	Clocks
mem48	5	4 NP

Flags

None.

SHLD - Double Precision Shift Left

0F A4	SHLD r/m16,r16,imm8	Shift r/m16 to left imm8 places while shifting bits from r16 in from the right
0F A5	SHLD r/m16,r16,CL	Shift r/m16 to left CL places while shifting bits from r16 in from the right
0F A4	SHLD r/m32,r32,imm8	Shift r/m32 to left imm8 places while shifting bits from r32 in from the right
0F A5	SHLD r/m32,r32,CL	Shift r/m32 to left CL places while shifting bits from r32 in from the right

Description

Shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHLD instruction is useful for multiprecision shifts of 64 bits or more.

Operands	Bytes	Clocks	
reg, reg, imm	4	4	NP
mem, reg, imm	4+d(0-2)	4	NP
reg, reg, cl	4	4	NP
mem, reg, cl	4+d(0-2)	5	NP

Flags

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

SHRD - Double Precision Shift Right

0F AC	SHRD r/m16,r16,imm8	Shift r/m16 to right imm8 places while shifting bits from r16 in from the left
0F AD	SHRD r/m16,r16,CL	Shift r/m16 to right CL places while shifting bits from r16 in from the left
0F AC	SHRD r/m32,r32,imm8	Shift r/m32 to right imm8 places while shifting bits from r32 in from the left
0F AD	SHRD r/m32,r32,CL	Shift r/m32 to right CL places while shifting bits from r32 in from the left

Description

Shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHRD instruction is useful for multiprecision shifts of 64 bits or more.

Operands	Bytes	Clocks	
reg, reg, imm	4	4	NP
mem, reg, imm	4+d(0-2)	4	NP
reg, reg, cl	4	4	NP
mem, reg, cl	4+d(0-2)	5	NP

Flags

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

SIDT - Store Interrupt Descriptor Table Register

See [SGDT/SIDT](#)

SLDT - Store Local Descriptor Table Register

0F 00 /0	SLDT r/m16	Stores segment selector from LDTR in r/m16
0F 00 /0	SLDT r/m32	Store segment selector from LDTR in low-order 16 bits of r/m32

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower-order 16 bits of the register. The high-order 16 bits of the register are cleared to 0s for the Pentium Pro processor and are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

The SLDT instruction is only useful in operating-system software; however, it can be used in application programs.

Operands	Bytes	Clocks
r16	3	2 NP
mem16	3+d(0-2)	2 NP

Flags

None.

SMSW - Store Machine Status Word

0F 01 /4	SMSW r/m16	Store machine status word to r/m16
0F 01 /4	SMSW r32/m16	Store machine status word in low-order 16 bits of r32/m16; high-order 16 bits of r32 are undefined

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a 16-bit general-purpose register or a memory location.

When the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the upper 16 bits of the register are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

The SMSW instruction is only useful in operating-system software; however, it is not a privileged instruction and can be used in application programs.

This instruction is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium Pro, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

Operands	Bytes	Clocks
r16	3	4 NP
mem16	3+d(0-2)	4 NP

Flags

None.

STC - Set Carry Flag
F9 STC Set CF flag

Description
Sets the CF flag in the EFLAGS register.

Operands	Bytes	Clocks
	1	2 NP

Flags
The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

STD - Set Direction Flag

FD STD Set DF flag

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

Operands	Bytes	Clocks
	1	2 NP

Flags

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

STI - Set Interrupt Flag

FB STI Set interrupt flag; external, maskable interrupts enabled
at the end of the next instruction

Description

Sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized 1. This behavior allows external interrupts to be disabled at the beginning of a procedure and enabled again at the end of the procedure. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions have no affect on the generation of exceptions and NMI interrupts.

The following decision table indicates the action of the STI instruction (bottom of the table) depending on the processor's mode of operation and the CPL and IOPL of the currently running program or procedure (top of the table).

PE =	0	1	1	1
VM =	X	0	0	1
CPL	X	\leq IOPL	$>$ IOPL	=3
IOPL	X	X	X	=3
IF \leftarrow 1	Y	Y	N	Y
#GP(0)	N	N	Y	N

NOTES:

X Don't care.

N Action in Column 1 not taken.

Y Action in Column 1 taken.

Operands	Bytes	Clocks
	1	7 NP

Flags

The IF flag is set to 1.

STOS/STOSB/STOSW/STOSD - Store String

AA	STOS m8	Store AL at address ES:(E)DI
AB	STOS m16	Store AX at address ES:(E)DI
AB	STOS m32	Store EAX at address ES:(E)DI
AA	STOSB	Store AL at address ES:(E)DI
AB	STOSW	Store AX at address ES:(E)DI
AB	STOSD	Store EAX at address ES:(E)DI

Description

Stores a byte, word, or doubleword from the AL, AX, or EAX register, respectively, into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the store string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and the AL, AX, or EAX register is assumed to be the source operand. The size of the destination and source operands is selected with the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), or STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the AL, AX, or EAX register to the memory location, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The STOS, STOSB, STOSW, and STOSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ/REPNE /REPZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Variations	Bytes	Clocks
stosb	1	3 NP
stosw	1	3 NP
stosd	1	3 NP
rep stosb	2	3+n NP
rep stosw	2	3+n NP
rep stosd	2	3+n NP

(n = count of bytes, words or dwords)

Flags

None.

STR - Store Task Register

0F 00 /1

STR r/m16

Stores segment selector from TR in r/m16

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared to 0s. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operands	Bytes	Clocks
r16	3	2 NP
mem16	3+d(0-2)	2 NP

Flags

None.

SUB - Subtract

2C ib	SUB AL, imm8	Subtract imm8 from AL
2D iw	SUB AX, imm16	Subtract imm16 from AX
2D id	SUB EAX, imm32	Subtract imm32 from EAX
80 /5 ib	SUB r/m8, imm8	Subtract imm8 from r/m8
81 /5 iw	SUB r/m16, imm16	Subtract imm16 from r/m16
81 /5 id	SUB r/m32, imm32	Subtract imm32 from r/m32
83 /5 ib	SUB r/m16, imm8	Subtract sign-extended imm8 from r/m16
83 /5 id	SUB r/m32, imm8	Subtract sign-extended imm8 from r/m32
28 / r	SUB r/m8, r8	Subtract r8 from r/m8
29 / r	SUB r/m16, r16	Subtract r16 from r/m16
29 / r	SUB r/m32, r32	Subtract r32 from r/m32
2A / r	SUB r8, r/m8	Subtract r/m8 from r8
2B / r	SUB r16, r/m16	Subtract r/m16 from r16
2B / r	SUB r32, r/m32	Subtract r/m32 from r32

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0,2)	3 UV
reg, mem	2+d(0,2)	2 UV
reg, imm	2+i(1,2)	1 UV
mem, imm	2+d(0,2)+i(1,2)	3 UV*
acc, imm	1+i(1,2)	1 UV

* = not pairable if there is a displacement and immediate

Flags

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

TEST - Logical Compare

A8 ib	TEST AL, imm8	AND imm8 with AL; set SF, ZF, PF according to result
A9 iw	TEST AX, imm16	AND imm16 with AX; set SF, ZF, PF according to result
A9 id	TEST EAX, imm32	AND imm32 with EAX; set SF, ZF, PF according to result
F6 /0 ib	TEST r/m8, imm8	AND imm8 with r/m8; set SF, ZF, PF according to result
F7 /0 iw	TEST r/m16, imm16	AND imm16 with r/m16; set SF, ZF, PF according to result
F7 /0 id	TEST r/m32, imm32	AND imm32 with r/m32; set SF, ZF, PF according to result
84 /r	TEST r/m8, r8	AND r8 with r/m8; set SF, ZF, PF according to result
85 /r	TEST r/m16, r16	AND r16 with r/m16; set SF, ZF, PF according to result
85 /r	TEST r/m32, r32	AND r32 with r/m32; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0,2)	2 UV
reg, mem	2+d(0,2)	2 UV
reg, imm	2+i(1,2)	1 UV
mem, imm	2+d(0,2)+i(1,2)	2 UV*
acc, imm	1+i(1,2)	1 UV

* = not pairable if there is a displacement and immediate

Flags

The OF and CF flags are cleared to 0. The SF, ZF, and PF flags are set according to the result (see the "Operation" section above). The state of the AF flag is undefined.

VERR/VERW - Verify a Segment for Reading or Writing

0F 00 /4 VERR r/m16 Set ZF=1 if segment specified with r/m16 can be read
0F 00 /5 VERW r/m16 Set ZF=1 if segment specified with r/m16 can be written

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not null.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

Operands	Bytes	Clocks
r16	3	7 NP
mem16	3+d(0,2)	7 NP

Flags

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is cleared to 0.

WAIT - Wait

9B	WAIT	Check pending unmasked floating-point exceptions.
9B	FWAIT	Check pending unmasked floating-point exceptions.

Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for the WAIT).

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction insures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 7 of the Intel Architecture Software Developer's Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

Operands	Bytes	Clocks
	1	1 NP

FPU Flags

The C0, C1, C2, and C3 flags are undefined.

WBINVD - Write Back and Invalidate Cache

0F 09 WBINVD Write back and flush Internal caches; initiate writing-back and flushing of external caches.

Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 7 of the Intel Architecture Software Developer's Manual, Volume 3).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

Operands	Bytes	Clocks
	2	2000+ NP

Flags

None.

WRMSR - Write to Model Specific Register

0F 30 WRMSR Write the value in EDX:EAX to MSR specified by ECX

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The high-order 32 bits are copied from EDX and the low-order 32 bits are copied from EAX. Always set the undefined or reserved bits in an MSR to the values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated, including the global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the Intel Architecture Software Developer's Manual, Volume 3). (MTRRs are an implementation-specific feature of the Pentium Pro processor.)

The MSRs control functions for testability, execution tracing, performance monitoring and machine check errors. Appendix B, Model-Specific Registers (MSRs), in the Intel Architecture Software Developer's Manual, Volume 3, lists all the MSRs that can be written to with this instruction and their addresses.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 7 of the Intel Architecture Software Developer's Manual, Volume 3).

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

Operands	Bytes	Clocks
	2	30-45 NP

Flags
None.

XADD - Exchange and Add

0F C0/r XADD r/m8,r8 Exchange r8 and r/m8; load sum into r/m8.
0F C1/r XADD r/m16,r16 Exchange r16 and r/m16; load sum into r/m16.
0F C1/r XADD r/m32,r32 Exchange r32 and r/m32; load sum into r/m32.

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

This instruction can be used with a LOCK prefix.

Operands	Bytes	Clocks
reg, reg	3	3 NP
mem, reg	3+d(0-2)	4 NP

Flags

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

XCHG - Exchange Register/Memory with Register

90+ rw	XCHG AX, r16	Exchange r16 with AX
90+ rw	XCHG r16, AX	Exchange AX with r16
90+ rd	XCHG EAX, r32	Exchange r32 with EAX
90+ rd	XCHG r32, EAX	Exchange EAX with r32
86 / r	XCHG r/m8, r8	Exchange r8 (byte register) with byte from r/m8
86 / r	XCHG r8, r/m8	Exchange byte from r/m8 with r8 (byte register)
87 / r	XCHG r/m16, r16	Exchange r16 with word from r/m16
87 / r	XCHG r16, r/m16	Exchange word from r/m16 with r16
87 / r	XCHG r/m32, r32	Exchange r32 with doubleword from r/m32
87 / r	XCHG r32, r/m32	Exchange doubleword from r/m32 with r32

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 7 of the Intel Architecture Software Developer's Manual, Volume 3, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

Operands	Bytes	Clocks
reg, reg	2	3 NP
reg, mem	2+d(0-2)	3 NP
mem, reg	2+d(0-2)	3 NP
acc, reg	1	2 NP
reg, acc	1	2 NP
in above: acc = AX or EAX only		

Flags

None.

XLAT/XLATB - Table Look-up Translation

D7	XLAT m8	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	Set AL to memory byte DS:[(E)BX + unsigned AL]

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operand" form and the "no-operand" form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

Operands	Bytes	Clocks
	1	4 NP

Flags

None.

XOR - Logical Exclusive OR

34 ib	XOR AL, imm8 AL	XOR imm8
35 iw	XOR AX, imm16 AX	XOR imm16
35 id	XOR EAX, imm32 EAX	XOR imm32
80 /6 ib	XOR r/m8, imm8 r/m8	XOR imm8
81 /6 iw	XOR r/m16, imm16 r/m16	XOR imm16
81 /6 id	XOR r/m32, imm32 r/m32	XOR imm32
83 /6 ib	XOR r/m16, imm8 r/m16	XOR imm8 (sign-extended)
83 /6 id	XOR r/m32, imm8 r/m32	XOR imm8 (sign-extended)
30 / r	XOR r/m8, r8 r/m8	XOR r8
31 / r	XOR r/m16, r16 r/m16	XOR r16
31 / r	XOR r/m32, r32 r/m32	XOR r32
32 / r	XOR r8, r/m8 r8	XOR r/m8
33 / r	XOR r16, r/m16 r8	XOR r/m8
33 / r	XOR r32, r/m32 r8	XOR r/m8

Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) 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.

Operands	Bytes	Clocks
reg, reg	2	1 UV
mem, reg	2+d(0,2)	3 UV
reg, mem	2+d(0,2)	2 UV
reg, imm	2+i(1,2)	1 UV
mem, imm	2+d(0,2)+i(1,2)	3 UV*
acc, imm	1+i(1,2)	1 UV

* = not pairable if there is a displacement and immediate

Flags

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

