



























































































































































































































































































































































































































































Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS (selector)		X	X	After a stack switch, a memory address exceeded the stack segment limit or was non-canonical.
		X	X	As part of a stack switch, the SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
		X		The IOPL was less than 3 and CR4.VME was 0.
		X		IOPL was less than 3, CR4.VME was 1, and the corresponding bit in the VME interrupt redirection bitmap was 1.
General protection, #GP (selector)	X	X	X	The interrupt vector was beyond the limit of IDT.
		X	X	The descriptor in the IDT was not an interrupt, trap, or task gate in legacy mode or not a 64-bit interrupt or trap gate in long mode.
		X	X	The DPL of the interrupt, trap, or task gate descriptor was less than the CPL.
		X	X	The segment selector specified by the interrupt or trap gate had its TI bit set, but the LDT selector was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate exceeded the descriptor table limit or was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate was not a code segment in legacy mode, or not a 64-bit code segment in long mode.
			X	The DPL of the segment specified by the interrupt or trap gate was greater than the CPL.
		X		The DPL of the segment specified by the interrupt or trap gate pointed was not 0 or it was a conforming segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## INTO

## Interrupt to Overflow Vector

Checks the overflow flag (OF) in the rFLAGS register and calls the overflow exception (#OF) handler if the OF flag is set to 1. This instruction has no effect if the OF flag is cleared to 0. The INTO instruction detects overflow in signed number addition. See *AMD64 Architecture Programmer’s Manual Volume 1: Application Programming* for more information on the OF flag.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

For detailed descriptions of the steps performed by INT instructions, see the following:

- *Legacy-Mode Interrupts*: “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts*: “Long-Mode Interrupt Control Transfers” in Volume 2.

Mnemonic	Opcode	Description
INTO	CE	Call overflow exception if the overflow flag is set. (Invalid in 64-bit mode.)

### Action

```

IF (64BIT_MODE)
    EXCEPTION [#UD]
IF (RFLAGS.OF == 1)      // #OF is a trap, and pushes the RIP of the instruction
    EXCEPTION [#OF]      // following INTO.
EXIT

```

### Related Instructions

INT, INT 3, BOUND

### rFLAGS Affected

None.

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Overflow, #OF	X	X	X	The INTO instruction was executed with OF set to 1.
Invalid opcode, #UD			X	Instruction was executed in 64-bit mode.

**Jcc****Jump on Condition**

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified by the condition code in the mnemonic (*cc*), jumps to the target instruction located at the specified relative offset. Otherwise, execution continues with the instruction following the Jcc instruction.

Unlike the unconditional jump (JMP), conditional jump instructions have only two forms—*short* and *near conditional jumps*. Different opcodes correspond to different forms of one instruction. For example, the JO instruction (jump if overflow) has opcode 0Fh 80h for its near form and 70h for its short form, but the mnemonic is the same for both forms. The only difference is that the near form has a 16- or 32-bit relative displacement, while the short form always has an 8-bit relative displacement.

Mnemonics are provided to deal with the programming semantics of both signed and unsigned numbers. Instructions tagged A (above) and B (below) are intended for use in unsigned integer code; those tagged G (greater) and L (less) are intended for use in signed integer code.

If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size.

In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit or 32-bit displacement value to 64 bits before adding it to the RIP.

These instructions cannot perform far jumps (to other code segments). To create a far-conditional-jump code sequence corresponding to a high-level language statement like:

```
IF A == B THEN GOTO FarLabel
```

where FarLabel is located in another code segment, use the opposite condition in a conditional short jump before an unconditional far jump. Such a code sequence might look like:

```
cmp    A,B          ; compare operands
jne   NextInstr     ; continue program if not equal
jmp   far FarLabel  ; far jump if operands are equal

NextInstr:           ; continue program
```

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JO rel8off	70 cb	
JO rel16off	0F 80 cw	Jump if overflow (OF = 1).
JO rel32off	0F 80 cd	
JNO rel8off	71 cb	
JNO rel16off	0F 81 cw	Jump if not overflow (OF = 0).
JNO rel32off	0F 81 cd	
JB rel8off	72 cb	
JB rel16off	0F 82 cw	Jump if below (CF = 1).
JB rel32off	0F 82 cd	

Mnemonic	Opcode	Description
JC <i>rel8off</i>	72 <i>cb</i>	
JC <i>rel16off</i>	0F 82 <i>cw</i>	Jump if carry (CF = 1).
JC <i>rel32off</i>	0F 82 <i>cd</i>	
JNAE <i>rel8off</i>	72 <i>cb</i>	
JNAE <i>rel16off</i>	0F 82 <i>cw</i>	Jump if not above or equal (CF = 1).
JNAE <i>rel32off</i>	0F 82 <i>cd</i>	
JNB <i>rel8off</i>	73 <i>cb</i>	
JNB <i>rel16off</i>	0F 83 <i>cw</i>	Jump if not below (CF = 0).
JNB <i>rel32off</i>	0F 83 <i>cd</i>	
JNC <i>rel8off</i>	73 <i>cb</i>	
JNC <i>rel16off</i>	0F 83 <i>cw</i>	Jump if not carry (CF = 0).
JNC <i>rel32off</i>	0F 83 <i>cd</i>	
JAE <i>rel8off</i>	73 <i>cb</i>	
JAE <i>rel16off</i>	0F 83 <i>cw</i>	Jump if above or equal (CF = 0).
JAE <i>rel32off</i>	0F 83 <i>cd</i>	
JZ <i>rel8off</i>	74 <i>cb</i>	
JZ <i>rel16off</i>	0F 84 <i>cw</i>	Jump if zero (ZF = 1).
JZ <i>rel32off</i>	0F 84 <i>cd</i>	
JE <i>rel8off</i>	74 <i>cb</i>	
JE <i>rel16off</i>	0F 84 <i>cw</i>	Jump if equal (ZF = 1).
JE <i>rel32off</i>	0F 84 <i>cd</i>	
JNZ <i>rel8off</i>	75 <i>cb</i>	
JNZ <i>rel16off</i>	0F 85 <i>cw</i>	Jump if not zero (ZF = 0).
JNZ <i>rel32off</i>	0F 85 <i>cd</i>	
JNE <i>rel8off</i>	75 <i>cb</i>	
JNE <i>rel16off</i>	0F 85 <i>cw</i>	Jump if not equal (ZF = 0).
JNE <i>rel32off</i>	0F 85 <i>cd</i>	
JBE <i>rel8off</i>	76 <i>cb</i>	
JBE <i>rel16off</i>	0F 86 <i>cw</i>	Jump if below or equal (CF = 1 or ZF = 1).
JBE <i>rel32off</i>	0F 86 <i>cd</i>	
JNA <i>rel8off</i>	76 <i>cb</i>	
JNA <i>rel16off</i>	0F 86 <i>cw</i>	Jump if not above (CF = 1 or ZF = 1).
JNA <i>rel32off</i>	0F 86 <i>cd</i>	
JNBE <i>rel8off</i>	77 <i>cb</i>	
JNBE <i>rel16off</i>	0F 87 <i>cw</i>	Jump if not below or equal (CF = 0 and ZF = 0).
JNBE <i>rel32off</i>	0F 87 <i>cd</i>	
JA <i>rel8off</i>	77 <i>cb</i>	
JA <i>rel16off</i>	0F 87 <i>cw</i>	Jump if above (CF = 0 and ZF = 0).
JA <i>rel32off</i>	0F 87 <i>cd</i>	
JS <i>rel8off</i>	78 <i>cb</i>	
JS <i>rel16off</i>	0F 88 <i>cw</i>	Jump if sign (SF = 1).
JS <i>rel32off</i>	0F 88 <i>cd</i>	
JNS <i>rel8off</i>	79 <i>cb</i>	
JNS <i>rel16off</i>	0F 89 <i>cw</i>	Jump if not sign (SF = 0).
JNS <i>rel32off</i>	0F 89 <i>cd</i>	

Mnemonic	Opcode	Description
JP rel8off	7A cb	
JP rel16off	0F 8A cw	Jump if parity (PF = 1).
JP rel32off	0F 8A cd	
JPE rel8off	7A cb	
JPE rel16off	0F 8A cw	Jump if parity even (PF = 1).
JPE rel32off	0F 8A cd	
JNP rel8off	7B cb	
JNP rel16off	0F 8B cw	Jump if not parity (PF = 0).
JNP rel32off	0F 8B cd	
JPO rel8off	7B cb	
JPO rel16off	0F 8B cw	Jump if parity odd (PF = 0).
JPO rel32off	0F 8B cd	
JL rel8off	7C cb	
JL rel16off	0F 8C cw	Jump if less (SF <> OF).
JL rel32off	0F 8C cd	
JNGE rel8off	7C cb	
JNGE rel16off	0F 8C cw	Jump if not greater or equal (SF <> OF).
JNGE rel32off	0F 8C cd	
JNL rel8off	7D cb	
JNL rel16off	0F 8D cw	Jump if not less (SF = OF).
JNL rel32off	0F 8D cd	
JGE rel8off	7D cb	
JGE rel16off	0F 8D cw	Jump if greater or equal (SF = OF).
JGE rel32off	0F 8D cd	
JLE rel8off	7E cb	
JLE rel16off	0F 8E cw	Jump if less or equal (ZF = 1 or SF <> OF).
JLE rel32off	0F 8E cd	
JNG rel8off	7E cb	
JNG rel16off	0F 8E cw	Jump if not greater (ZF = 1 or SF <> OF).
JNG rel32off	0F 8E cd	
JNLE rel8off	7F cb	
JNLE rel16off	0F 8F cw	Jump if not less or equal (ZF = 0 and SF = OF).
JNLE rel32off	0F 8F cd	
JG rel8off	7F cb	
JG rel16off	0F 8F cw	Jump if greater (ZF = 0 and SF = OF).
JG rel32off	0F 8F cd	

## Related Instructions

JMP (Near), JMP (Far), JrCXZ

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

**JCXZ****Jump if rCX Zero****JECXZ****JRCXZ**

Checks the contents of the count register (rCX) and, if 0, jumps to the target instruction located at the specified 8-bit relative offset. Otherwise, execution continues with the instruction following the JrCXZ instruction.

The size of the count register (CX, ECX, or RCX) depends on the address-size attribute of the JrCXZ instruction. Therefore, JRCXZ can only be executed in 64-bit mode and JCXZ cannot be executed in 64-bit mode.

If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size.

In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit displacement value to 64 bits before adding it to the RIP.

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JCXZ <i>rel8off</i>	E3 cb	Jump short if the 16-bit count register (CX) is zero.
JECXZ <i>rel8off</i>	E3 cb	Jump short if the 32-bit count register (ECX) is zero.
JRCXZ <i>rel8off</i>	E3 cb	Jump short if the 64-bit count register (RCX) is zero.

**Related Instructions**

Jcc, JMP (Near), JMP (Far)

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical

## JMP (Near)

## Near Jump

Unconditionally transfers control to a new address without saving the current rIP value. This form of the instruction jumps to an address in the current code segment and is called a *near jump*. The target operand can specify a register, a memory location, or a label.

If the JMP target is specified in a register or memory location, then a 16-, 32-, or 64-bit rIP is read from the operand, depending on operand size. This rIP is zero-extended to 64 bits.

If the JMP target is specified by a displacement in the instruction, the signed displacement is added to the rIP (of the following instruction), and the result is truncated to 16, 32, or 64 bits depending on operand size. The signed displacement can be 8 bits, 16 bits, or 32 bits, depending on the opcode and the operand size.

For near jumps in 64-bit mode, the operand size defaults to 64 bits. The E9 opcode results in  $\text{RIP} = \text{RIP} + 32\text{-bit signed displacement}$ , and the FF /4 opcode results in  $\text{RIP} = 64\text{-bit offset from register or memory}$ . No prefix is available to encode a 32-bit operand size in 64-bit mode.

See JMP (Far) for information on far jumps—jumps to procedures located outside of the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JMP <i>rel8off</i>	EB <i>cb</i>	Short jump with the target specified by an 8-bit signed displacement.
JMP <i>rel16off</i>	E9 <i>cw</i>	Near jump with the target specified by a 16-bit signed displacement.
JMP <i>rel32off</i>	E9 <i>cd</i>	Near jump with the target specified by a 32-bit signed displacement.
JMP <i>reg/mem16</i>	FF /4	Near jump with the target specified <i>reg/mem16</i> .
JMP <i>reg/mem32</i>	FF /4	Near jump with the target specified <i>reg/mem32</i> . (No prefix for encoding in 64-bit mode.)
JMP <i>reg/mem64</i>	FF /4	Near jump with the target specified <i>reg/mem64</i> .

## Related Instructions

JMP (Far), Jcc, JrCX

## rFLAGS Affected

None.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## JMP (Far)

## Far Jump

Unconditionally transfers control to a new address without saving the current CS:rIP values. This form of the instruction jumps to an address outside the current code segment and is called a *far jump*. The operand specifies a target selector and offset.

The target operand can be specified by the instruction directly, by containing the far pointer in the jmp far opcode itself, or indirectly, by referencing a far pointer in memory. In 64-bit mode, only indirect far jumps are allowed, executing a direct far jmp (opcode EA) will generate an undefined opcode exception. For both direct and indirect far jumps, if the JMP (Far) operand-size is 16 bits, the instruction's operand is a 16-bit selector followed by a 16-bit offset. If the operand-size is 32 or 64 bits, the operand is a 16-bit selector followed by a 32-bit offset.

In all modes, the target selector used by the instruction can be a code selector. Additionally, the target selector can also be a call gate in protected mode, or a task gate or TSS selector in legacy protected mode.

- *Target is a code segment*—Control is transferred to the target CS:rIP. In this case, the target offset can only be a 16 or 32 bit value, depending on operand-size, and is zero-extended to 64 bits. No CPL change is allowed.
- *Target is a call gate*—The call gate specifies the actual target code segment and offset, and control is transferred to the target CS:rIP. When jumping through a call gate, the size of the target rIP is 16, 32, or 64 bits, depending on the size of the call gate. If the target rIP is less than 64 bits, it's zero-extended to 64 bits. In long mode, only 64-bit call gates are allowed, and they must point to 64-bit code segments. No CPL change is allowed.
- *Target is a task gate or a TSS*—If the mode is legacy protected mode, then a task switch occurs. See “Hardware Task-Management in Legacy Mode” in volume 2 for details about task switches. Hardware task switches are not supported in long mode.

See JMP (Near) for information on near jumps—jumps to procedures located inside the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JMP FAR <i>pntr16:16</i>	EA <i>cd</i>	Far jump direct, with the target specified by a far pointer contained in the instruction. (Invalid in 64-bit mode.)
JMP FAR <i>pntr16:32</i>	EA <i>cp</i>	Far jump direct, with the target specified by a far pointer contained in the instruction. (Invalid in 64-bit mode.)
JMP FAR <i>mem16:16</i>	FF /5	Far jump indirect, with the target specified by a far pointer in memory.
JMP FAR <i>mem16:32</i>	FF /5	Far jump indirect, with the target specified by a far pointer in memory.

## Action

```
// Far jumps (JMPF)
// See "Pseudocode Definition" on page 57.

JMPF_START:

IF (REAL_MODE)
    JMPF_REAL_OR_VIRTUAL
ELSIF (PROTECTED_MODE)
    JMPF_PROTECTED
ELSE // (VIRTUAL_MODE)
    JMPF_REAL_OR_VIRTUAL

JMPF_REAL_OR_VIRTUAL:

IF (OPCODE == jmpf [mem]) // JMPF Indirect
{
    temp_RIP = READ_MEM.z [mem]
    temp_CS = READ_MEM.w [mem+Z]
}
ELSE // (OPCODE == jmpf direct)
{
    temp_RIP = z-sized offset specified in the instruction,
               zero-extended to 64 bits
    temp_CS = selector specified in the instruction
}

IF (temp_RIP>CS.limit)
    EXCEPTION [#GP(0)]

CS.sel = temp_CS
CS.base = temp_CS SHL 4
RIP = temp_RIP
EXIT

JMPF_PROTECTED:

IF (OPCODE == jmpf [mem]) // JMPF Indirect
{
    temp_offset = READ_MEM.z [mem]
    temp_sel = READ_MEM.w [mem+Z]
}
ELSE // (OPCODE == jmpf direct)
{
    IF (64BIT_MODE)
        EXCEPTION [#UD]           // 'jmpf direct' is illegal in 64-bit mode

    temp_offset = z-sized offset specified in the instruction,
                  zero-extended to 64 bits
    temp_sel = selector specified in the instruction
}
```

```

temp_desc = READ_DESCRIPTOR (temp_sel, cs_chk)
                // read descriptor, perform protection and type checks

IF (temp_desc.attr.type == 'available_tss')
    TASK_SWITCH      // using temp_sel as the target tss selector
ELSIF (temp_desc.attr.type == 'taskgate')
    TASK_SWITCH      // using the tss selector in the task gate as the
                    // target tss
ELSIF (temp_desc.attr.type == 'code')
    // if the selector refers to a code descriptor, then
    // the offset we read is the target RIP
{
    temp_RIP = temp_offset
    CS = temp_desc
    IF ((!64BIT_MODE) && (temp_RIP > CS.limit))
        // temp_RIP can't be non-canonical because
        // it's a 16- or 32-bit offset, zero-extended to 64 bits
    {
        EXCEPTION [#GP(0)]
    }
    RIP = temp_RIP
    EXIT
}
ELSE
{
    // (temp_desc.attr.type == 'callgate')
    // if the selector refers to a call gate, then
    // the target CS and RIP both come from the call gate
    temp_RIP = temp_desc.offset

    IF (LONG_MODE)
    {
        // in long mode, we need to read the 2nd half of a 16-byte call-gate
        // from the gdt/ldt to get the upper 32 bits of the target RIP
        temp_upper = READ_MEM.q [temp_sel+8]
        IF (temp_upper's extended attribute bits != 0)
            EXCEPTION [#GP(temp_sel)]      // Make sure the extended
                                            // attribute bits are all zero.

        temp_RIP = tempRIP + (temp_upper SHL 32)
                // concatenate both halves of RIP
    }
    CS = READ_DESCRIPTOR (temp_desc.segment, clg_chk)
                // set up new CS base, attr, limits
    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
    || (!64BIT_MODE) && (temp_RIP > CS.limit))
        EXCEPTION [#GP(0)]
    RIP = temp_RIP
    EXIT
}

```

## Related Instructions

JMP (Near), Jcc, JrCX

## rFLAGS Affected

None, unless a task switch occurs, in which case all flags are modified.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The far JUMP indirect opcode (FF /5) had a register operand.
			X	The far JUMP direct opcode (EA) was executed in 64-bit mode.
Segment not present, #NP (selector)			X	The accessed code segment, call gate, task gate, or TSS was not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)		X		The target code segment selector was a null selector.
		X		A code, call gate, task gate, or TSS descriptor exceeded the descriptor table limit.
		X		A segment selector's TI bit was set, but the LDT selector was a null selector.
		X		The segment descriptor specified by the instruction was not a code segment, task gate, call gate or available TSS in legacy mode, or not a 64-bit code segment or a 64-bit call gate in long mode.
		X		The RPL of the non-conforming code segment selector specified by the instruction was greater than the CPL, or its DPL was not equal to the CPL.
		X		The DPL of the conforming code segment descriptor specified by the instruction was greater than the CPL.
		X		The DPL of the callgate, taskgate, or TSS descriptor specified by the instruction was less than the CPL or less than its own RPL.
		X		The segment selector specified by the call gate or task gate was a null selector.
		X		The segment descriptor specified by the call gate was not a code segment in legacy mode or not a 64-bit code segment in long mode.
		X		The DPL of the segment descriptor specified by the call gate was greater than the CPL and it is a conforming segment.
		X		The DPL of the segment descriptor specified by the callgate was not equal to the CPL and it is a non-conforming segment.
		X		The 64-bit call gate's extended attribute bits were not zero.
		X		The TSS descriptor was found in the LDT.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## LAHF

## Load Status Flags into AH Register

Loads the lower 8 bits of the rFLAGS register, including sign flag (SF), zero flag (ZF), auxiliary carry flag (AF), parity flag (PF), and carry flag (CF), into the AH register.

The instruction sets the reserved bits 1, 3, and 5 of the rFLAGS register to 1, 0, and 0, respectively, in the AH register.

The LAHF instruction can only be executed in 64-bit mode. Support for the instruction is implementation-dependent and is indicated by CPUID Fn8000\_0001(ECX[LahfSahf] = 1).

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
LAHF	9F	Load the SF, ZF, AF, PF, and CF flags into the AH register.

### Related Instructions

SAHF

### rFLAGS Affected

None.

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	The LAHF instruction is not supported, as indicated by CPUID Fn8000_0001(ECX[LahfSahf] = 0).

## LDS LES LFS LGS LSS

## Load Far Pointer

Loads a far pointer from a memory location (second operand) into a segment register (mnemonic) and general-purpose register (first operand). The instruction stores the 16-bit segment selector of the pointer into the segment register and the 16-bit or 32-bit offset portion into the general-purpose register. The operand-size attribute determines whether the pointer loaded is 32 or 48 bits in length.

These instructions load associated segment-descriptor information into the hidden portion of the specified segment register.

Mnemonic	Opcode	Description
LDS <i>reg16, mem16:16</i>	C5 /r	Load DS:reg16 with a far pointer from memory. [Redefined as VEX (2-byte prefix) in 64-bit mode.]
LDS <i>reg32, mem16:32</i>	C5 /r	Load DS:reg32 with a far pointer from memory. [Redefined as VEX (2-byte prefix) in 64-bit mode.]
LES <i>reg16, mem16:16</i>	C4 /r	Load ES:reg16 with a far pointer from memory. [Redefined as VEX (3-byte prefix) in 64-bit mode.]
LES <i>reg32, mem16:32</i>	C4 /r	Load ES:reg32 with a far pointer from memory. [Redefined as VEX (3-byte prefix) in 64-bit mode.]
LFS <i>reg16, mem16:16</i>	0F B4 /r	Load FS:reg16 with a 32-bit far pointer from memory.
LFS <i>reg32, mem16:32</i>	0F B4 /r	Load FS:reg32 with a 48-bit far pointer from memory.
LGS <i>reg16, mem16:16</i>	0F B5 /r	Load GS:reg16 with a 32-bit far pointer from memory.
LGS <i>reg32, mem16:32</i>	0F B5 /r	Load GS:reg32 with a 48-bit far pointer from memory.
LSS <i>reg16, mem16:16</i>	0F B2 /r	Load SS:reg16 with a 32-bit far pointer from memory.
LSS <i>reg32, mem16:32</i>	0F B2 /r	Load SS:reg32 with a 48-bit far pointer from memory.

## Related Instructions

None

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The source operand was a register.
			X	LDS or LES was executed in 64-bit mode.
Segment not present, #NP (selector)			X	The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	A segment register was loaded, but the segment descriptor exceeded the descriptor table limit.
			X	A segment register was loaded and the segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The SS register was loaded with a null segment selector in non-64-bit mode or while CPL = 3.
			X	The SS register was loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL.
			X	The SS register was loaded and the segment pointed to was not a writable data segment.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was a data or non-conforming code segment, but the RPL or CPL was greater than the DPL.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was not a data segment or readable code segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## LEA

## Load Effective Address

Computes the effective address of a memory location (second operand) and stores it in a general-purpose register (first operand).

The address size of the memory location and the size of the register determine the specific action taken by the instruction, as follows:

- If the address size and the register size are the same, the instruction stores the effective address as computed.
- If the address size is longer than the register size, the instruction truncates the effective address to the size of the register.
- If the address size is shorter than the register size, the instruction zero-extends the effective address to the size of the register.

If the second operand is a register, an undefined-opcode exception occurs.

The LEA instruction is related to the MOV instruction, which copies data from a memory location to a register, but LEA takes the address of the source operand, whereas MOV takes the contents of the memory location specified by the source operand. In the simplest cases, LEA can be replaced with MOV. For example:

```
lea eax, [ebx]
```

has the same effect as:

```
mov eax, ebx
```

However, LEA allows software to use any valid ModRM and SIB addressing mode for the source operand. For example:

```
lea eax, [ebx+edi]
```

loads the sum of the EBX and EDI registers into the EAX register. This could not be accomplished by a single MOV instruction.

The LEA instruction has a limited capability to perform multiplication of operands in general-purpose registers using scaled-index addressing. For example:

```
lea eax, [ebx+ebx*8]
```

loads the value of the EBX register, multiplied by 9, into the EAX register. Possible values of multipliers are 2, 4, 8, 3, 5, and 9.

The LEA instruction is widely used in string-processing and array-processing to initialize an index register (rSI or rDI) before performing string instructions such as MOVSx. It is also used to initialize the rBX register before performing the XLAT instruction in programs that perform character translations. In data structures, the LEA instruction can calculate addresses of operands stored in memory, and in particular, addresses of array or string elements.

Mnemonic	Opcode	Description
LEA reg16, mem	8D /r	Store effective address in a 16-bit register.
LEA reg32, mem	8D /r	Store effective address in a 32-bit register.
LEA reg64, mem	8D /r	Store effective address in a 64-bit register.

## Related Instructions

MOV

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The source operand was a register.

## LEAVE

## Delete Procedure Stack Frame

Releases a stack frame created by a previous ENTER instruction. To release the frame, it copies the frame pointer (in the rBP register) to the stack pointer register (rSP), and then pops the old frame pointer from the stack into the rBP register, thus restoring the stack frame of the calling procedure.

The 32-bit LEAVE instruction is equivalent to the following 32-bit operation:

```
MOV ESP, EBP
POP EBP
```

To return program control to the calling procedure, execute a RET instruction after the LEAVE instruction.

In 64-bit mode, the LEAVE operand size defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size.

Mnemonic	Opcode	Description
LEAVE	C9	Set the stack pointer register SP to the value in the BP register and pop BP.
LEAVE	C9	Set the stack pointer register ESP to the value in the EBP register and pop EBP. (No prefix for encoding this in 64-bit mode.)
LEAVE	C9	Set the stack pointer register RSP to the value in the RBP register and pop RBP.

### Related Instructions

ENTER

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## LFENCE

## Load Fence

Acts as a barrier to force strong memory ordering (serialization) between load instructions preceding the LFENCE and load instructions that follow the LFENCE. Loads from differing memory types may be performed out of order, in particular between WC/WC+ and other memory types. The LFENCE instruction assures that the system completes all previous loads before executing subsequent loads.

The LFENCE instruction is weakly-ordered with respect to store instructions, data and instruction prefetches, and the SFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an LFENCE.

In addition to load instructions, the LFENCE instruction is strongly ordered with respect to other LFENCE instructions, as well as MFENCE and other serializing instructions. Further details on the use of MFENCE to order accesses among differing memory types may be found in *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*, section 7.4 “Memory Types” on page 172.

LFENCE is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
LFENCE	0F AE E8	Force strong ordering of (serialize) load operations.

### Related Instructions

MFENCE, SFENCE

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.

## LLWPCB

## Load Lightweight Profiling Control Block Address

Parses the Lightweight Profiling Control Block at the address contained in the specified register. If the LWPCB is valid, writes the address into the LWP\_CBAADDR MSR and enables Lightweight Profiling.

See Volume 2, Chapter 13, for an overview of the lightweight profiling facility.

The LWPCB must be in memory that is readable and writable in user mode. For better performance, it should be aligned on a 64-byte boundary in memory and placed so that it does not cross a page boundary, though neither of these suggestions is required.

The LWPCB address in the register is truncated to 32 bits if the operand size is 32.

### Action

1. If LWP is not available or if the machine is not in protected mode, LLWPCB immediately causes a #UD exception.
2. If LWP is already enabled, the processor flushes the LWP state to memory in the old LWPCB. See description of the SLWPCB instruction on page 326 for details on saving the active LWP state.  
If the flush causes a #PF exception, LWP remains enabled with the old LWPCB still active. Note that the flush is done before LWP attempts to access the new LWPCB.
3. If the specified LWPCB address is 0, LWP is disabled and the execution of LLWPCB is complete.
4. The LWPCB address is non-zero. LLWPCB validates it as follows:
  - If any part of the LWPCB or the ring buffer is beyond the data segment limit, LLWPCB causes a #GP exception.
  - If the ring buffer size is below the implementation's minimum ring buffer size, LLWPCB causes a #GP exception.
  - While doing these checks, LWP reads and writes the LWPCB, which may cause a #PF exception.

If any of these exceptions occurs, LLWPCB aborts and LWP is left disabled. Usually, the operating system will handle a #PF exception by making the memory available and returning to retry the LLWPCB instruction. The #GP exceptions indicate application programming errors.

5. LWP converts the LWPCB address and the ring buffer address to linear address form by adding the DS base address and stores the addresses internally.
6. LWP examines the LWPCB.Flags field to determine which events should be enabled and whether threshold interrupts should be taken. It clears the bits for any features that are not available and stores the result back to LWPCB.Flags to inform the application of the actual LWP state.
7. For each event being enabled, LWP examines the EventInterval $n$  value and, if necessary, sets it to an implementation-defined minimum. (The minimum event interval for LWPVAL is zero.) It loads its internal counter for the event from the value in EventCountern. A zero or negative value

in EventCountn means that the next event of that type will cause an event record to be stored. To count every  $j^{\text{th}}$  event, a program should set EventIntervaln to  $j-1$  and EventCountn to some starting value (where  $j-1$  is a good initial count). If the counter value is larger than the interval, the first event record will be stored after a larger number of events than subsequent records.

8. LWP is started. The execution of LLWPCB is complete.

## Notes

If none of the bits in the LWPCB.Flags specifies an available event, LLWPCB still enables LWP to allow the use of the LWPINS instruction. However, no other event records will be stored.

A program can temporarily disable LWP by executing SLWPCB to obtain the current LWPCB address, saving that value, and then executing LLWPCB with a register containing 0. It can later re-enable LWP by executing LLWPCB with a register containing the saved address.

When LWP is enabled, it is typically an error to execute LLWPCB with the address of the active LWPCB. When the hardware flushes the existing LWP state into the LWPCB, it may overwrite fields that the application may have set to new LWP parameter values. The flushed values will then be loaded as LWP is restarted. To reuse an LWPCB, an application should stop LWP by passing a zero to LLWPCB, then prepare the LWPCB with new parameters and execute LLWPCB again to restart LWP.

Internally, LWP keeps the linear address of the LWPCB and the ring buffer. If the application changes the value of DS, LWP will continue to collect samples even if the new DS value would no longer allow access the LWPCB or the ring buffer. However, a #GP fault will occur if the application uses XRSTOR to restore LWP state saved by XSAVE. Programs should avoid using XSAVE/XRSTOR on LWP state if DS has changed. This only applies when the CPL != 0; kernel mode operation of XRSTOR is unaffected by changes to DS. See instruction listing for XSAVE in Volume 4 for details.

Operating system and hypervisor code that runs when CPL  $\neq 3$  should use XSAVE and XRSTOR to control LWP rather than using LLWPCB. Use WRMSR to write 0 to the LWP\_CBADDR MSR to immediately stop LWP without saving its current state.

It is possible to execute LLWPCB when the CPL != 3 or when SMM is active, but the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Furthermore, if LWP is enabled when a kernel executes LLWPCB, both the old and new control blocks and ring buffers must be accessible. Using LLWPCB in these situations is not recommended.

LLWPCB is an LWP instruction. Support for LWP instructions is indicated by CPUID Fn8000\_0001(ECX[LWP] = 1).

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

## Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
LLWPCB <i>reg32</i>	8F	$\overline{\text{RXB}}.09$	0.1111.0.00	12 /0
LLWPCB <i>reg64</i>	8F	$\overline{\text{RXB}}.09$	1.1111.0.00	12 /0

ModRM.reg augments the opcode and is assigned the value 0. ModRM.r/m (augmented by XOP.R) specifies the register containing the effective address of the LWPCB. ModRM.mod is 11b.

## Related Instructions

SLWPCB, LWPVAL, LWPINS

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP instructions are not supported, as indicated by CPUID Fn8000_0001(ECX[LWP]) = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available, or mod != 11b, or vvvv != 1111b.
General protection, #GP			X	Any part of the LWPCB or the event ring buffer is beyond the DS segment limit.
			X	Any restrictions on the contents of the LWPCB are violated
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	LWP was already enabled and a page fault resulted from reading or writing the old LWPCB.
			X	LWP was already enabled and a page fault resulted from flushing an event to the old ring buffer.

## **LODS LODSB LODSW LODSD LODSQ**

## **Load String**

Copies the byte, word, doubleword, or quadword in the memory location pointed to by the DS:rSI registers to the AL, AX, EAX, or RAX register, depending on the size of the operand, and then increments or decrements the rSI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements rSI by 1, 2, 4, or 8, depending on the number of bytes being loaded.

The forms of the LODS instruction with an explicit operand address the operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix. The explicit operand serves only to specify the type (size) of the value being copied and the specific registers used.

The no-operands forms of the instruction always use the DS:[rSI] registers to point to the value to be copied (they do not allow a segment prefix). The mnemonic determines the size of the operand and the specific registers used.

The LODSx instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 12. More often, software uses the LODSx instruction inside a loop controlled by a LOOPcc instruction as a more efficient replacement for instructions like:

```
mov eax, dword ptr ds:[esi]
add esi, 4
```

The LODSQ instruction can only be used in 64-bit mode.

Mnemonic	Opcode	Description
LODS <i>mem8</i>	AC	Load byte at DS:rSI into AL and then increment or decrement rSI.
LODS <i>mem16</i>	AD	Load word at DS:rSI into AX and then increment or decrement rSI.
LODS <i>mem32</i>	AD	Load doubleword at DS:rSI into EAX and then increment or decrement rSI.
LODS <i>mem64</i>	AD	Load quadword at DS:rSI into RAX and then increment or decrement rSI.
LODSB	AC	Load byte at DS:rSI into AL and then increment or decrement rSI.
LODSW	AD	Load the word at DS:rSI into AX and then increment or decrement rSI.

Mnemonic	Opcode	Description
LODSD	AD	Load doubleword at DS:rSI into EAX and then increment or decrement rSI.
LODSQ	AD	Load quadword at DS:rSI into RAX and then increment or decrement rSI.

## Related Instructions

MOV\$*x*, STOS*x*

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## **LOOP** **LOOPE** **LOOPNE** **LOOPNZ** **LOOPZ**

## **Loop**

Decrements the count register (rCX) by 1, then, if rCX is not 0 and the ZF flag meets the condition specified by the mnemonic, it jumps to the target instruction specified by the signed 8-bit relative offset. Otherwise, it continues with the next instruction after the LOOPcc instruction.

The size of the count register used (CX, ECX, or RCX) depends on the address-size attribute of the LOOPcc instruction.

The LOOP instruction ignores the state of the ZF flag.

The LOOPE and LOOPZ instructions jump if rCX is not 0 and the ZF flag is set to 1. In other words, the instruction exits the loop (falls through to the next instruction) if rCX becomes 0 or ZF = 0.

The LOOPNE and LOOPNZ instructions jump if rCX is not 0 and ZF flag is cleared to 0. In other words, the instruction exits the loop if rCX becomes 0 or ZF = 1.

The LOOPcc instruction does not change the state of the ZF flag. Typically, the loop contains a compare instruction to set or clear the ZF flag.

If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size.

In 64-bit mode, the operand size defaults to 64 bits without the need for a REX prefix, and the processor sign-extends the 8-bit offset before adding it to the RIP.

Mnemonic	Opcode	Description
LOOP <i>rel8off</i>	E2 cb	Decrement rCX, then jump short if rCX is not 0.
LOOPE <i>rel8off</i>	E1 cb	Decrement rCX, then jump short if rCX is not 0 and ZF is 1.
LOOPNE <i>rel8off</i>	E0 cb	Decrement rCX, then Jump short if rCX is not 0 and ZF is 0.
LOOPNZ <i>rel8off</i>	E0 cb	Decrement rCX, then Jump short if rCX is not 0 and ZF is 0.
LOOPZ <i>rel8off</i>	E1 cb	Decrement rCX, then Jump short if rCX is not 0 and ZF is 1.

## **Related Instructions**

None

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

## LWPINS

## Lightweight Profiling Insert Record

Inserts programmed event record into the LWP event ring buffer in memory and advances the ring buffer pointer.

Refer to the description of the programmed event record in Volume 2, Chapter 13. The record has an EventId of 255. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2.

If the ring buffer is not full, or if LWP is running in Continuous Mode, the head pointer is advanced and the CF flag is cleared. If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in Continuous Mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in Synchronized Mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, the head pointer is not advanced, and the CF flag is set.

LWPINS generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPINS simply clears CF if LWP is not enabled. This allows LWPINS instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPINS when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPINS in these situations is not recommended.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. For instance, a program might capture information on changes in the process' address space such as library loads and unloads, or changes in the execution environment such as a change in the state of a user-mode thread of control.

Note that when the LWPINS instruction finishes writing a event record in the event ring buffer, it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store another event record with the same instruction address (but different EventId values).

LWPINS is an LWP instruction. Support for LWP instructions is indicated by CPUID Fn8000\_0001(ECX[LWP]) = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

## Instruction Encoding

Mnemonic	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
LWPINS reg32.vvvv, reg/mem32, imm32	8F	$\overline{\text{RXB}}.0\text{A}$	0. $\overline{\text{src1}}$ .0.00	12 /0 /imm32
LWPINS reg64.vvvv, reg/mem32, imm32	8F	$\overline{\text{RXB}}.0\text{A}$	1. $\overline{\text{src1}}$ .0.00	12 /0 /imm32

ModRM.reg augments the opcode and is assigned the value 0. The {mod, r/m} field of the ModRM byte (augmented by XOP.R) encodes the second operand. A 4-byte immediate field follows ModRM.

## Related Instructions

LLWPCB, SLWPCB, LWPVAL

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[LWP] = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from writing the event to the ring buffer.
			X	A page fault resulted from reading a modrm operand from memory.
General protection, #GP			X	A modrm operand in memory exceeded the segment limit.

## LWPVAL

## Lightweight Profiling Insert Value

Decrements the event counter associated with the programmed value sample event (see “Programmed Value Sample” in Volume 2, Chapter 13). If the resulting counter value is negative, inserts an event record into the LWP event ring buffer in memory and advances the ring buffer pointer.

Refer to the description of the programmed value sample record in Volume 2, Chapter 13. The event record has an EventId of 1. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2.

If the programmed value sample record is not written to the event ring buffer, the memory location of the second operand (assuming it is memory-based) is not accessed.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the event counter is reset to the interval for the event (subject to randomization). If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in Continuous Mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in Synchronized Mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, and the head pointer is not advanced.

LWPVAL generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPVAL does nothing if LWP is not enabled or if the Programmed Value Sample event is not enabled in LWPCB.Flags. This allows LWPVAL instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPVAL when the CPL != 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPVAL in these situations is not recommended.

LWPVAL can be used by a program to perform value profiling. This is the technique of sampling the value of some program variable at a predetermined frequency. For example, a managed runtime might use LWPVAL to sample the value of the divisor for a frequently executed divide instruction in order to determine whether to generate specialized code for a common division. It might sample the target location of an indirect branch or call to see if one destination is more frequent than others. Since LWPVAL does not modify any registers or condition codes, it can be inserted harmlessly between any instructions.

## Note

When LWPVAL completes (whether or not it stored an event record in the event ring buffer), it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store an event record. If LWPVAL also stored an event record, the buffer will contain two records with the same instruction address (but different EventId values).

LWPVAL is an LWP instruction. Support for LWP instructions is indicated by CPUID Fn8000\_0001\_ECX[LWP] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

## Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
LWPVAL <i>reg32.vvvv, reg/mem32, imm32</i>	8F	$\overline{\text{RXB}}.0A$	0.src1.0.00	12 /1 /imm32
LWPVAL <i>reg64.vvvv, reg/mem32, imm32</i>	8F	$\overline{\text{RXB}}.0A$	1.src1.0.00	12 /1 /imm32

ModRM.reg augments the opcode and is assigned the value 001b. The {mod, r/m} field of the ModRM byte (augmented by XOP.R) encodes the second operand. A four-byte immediate field follows ModRM.

## Related Instructions

LLWPCB, SLWPCB, LWPINS

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP instructions are not supported, as indicated by CPUID Fn8000_0001(ECX[LWP]) = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from writing the event to the ring buffer.
			X	A page fault resulted from reading a modrm operand from memory.
General protection, #GP			X	A modrm operand in memory exceeded the segment limit.

## LZCNT

## Count Leading Zeros

Counts the number of leading zero bits in the 16-, 32-, or 64-bit general purpose register or memory source operand. Counting starts downward from the most significant bit and stops when the highest bit having a value of 1 is encountered or when the least significant bit is encountered. The count is written to the destination register.

This instruction has two operands:

*LZCNT dest, src*

If the input operand is zero, CF is set to 1 and the size (in bits) of the input operand is written to the destination register. Otherwise, CF is cleared.

If the most significant bit is a one, the ZF flag is set to 1, zero is written to the destination register. Otherwise, ZF is cleared.

LZCNT is an Advanced Bit Manipulation (ABM) instruction. Support for the LZCNT instruction is indicated by CPUID Fn8000\_0001\_ECX[ABM] = 1. If the LZCNT instruction is not available, the encoding is interpreted as the BSR instruction. Software MUST check the CPUID bit once per program or library initialization before using the LZCNT instruction, or inconsistent behavior may result.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
LZCNT <i>reg16, reg/mem16</i>	F3 0F BD /r	Count the number of leading zeros in reg/mem16.
LZCNT <i>reg32, reg/mem32</i>	F3 0F BD /r	Count the number of leading zeros in reg/mem32.
LZCNT <i>reg64, reg/mem64</i>	F3 0F BD /r	Count the number of leading zeros in reg/mem64.

### Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, POPCNT, T1MSKC, TZCNT, TZMSK

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## MFENCE

## Memory Fence

Acts as a barrier to force strong memory ordering (serialization) between load and store instructions preceding the MFENCE, and load and store instructions that follow the MFENCE. The processor may perform loads out of program order with respect to non-conflicting stores for certain memory types. The MFENCE instruction guarantees that the system completes all previous memory accesses before executing subsequent accesses.

The MFENCE instruction is weakly-ordered with respect to data and instruction prefetches. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an MFENCE.

In addition to load and store instructions, the MFENCE instruction is strongly ordered with respect to other MFENCE instructions, LFENCE instructions, SFENCE instructions, serializing instructions, and CLFLUSH instructions. Further details on the use of MFENCE to order accesses among differing memory types may be found in *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*, section 7.4 “Memory Types” on page 172.

The MFENCE instruction is a serializing instruction.

MFENCE is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

### Instruction Encoding

Mnemonic	Opcode	Description
MFENCE	0F AE F0	Force strong ordering of (serialized) load and store operations.

### Related Instructions

LFENCE, SFENCE

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.

## MONITORX

Establishes a linear address range of memory for hardware to monitor and puts the processor in the monitor event pending state. When in the monitor event pending state, the monitoring hardware detects stores to the specified linear address range and causes the processor to exit the monitor event pending state. The MWAIT and MWAITX instructions use the state of the monitor hardware.

The address range should be a write-back memory type. Executing MONITORX on an address range for a non-write-back memory type is not guaranteed to cause the processor to enter the monitor event pending state. The size of the linear address range that is established by the MONITORX instruction can be determined by CPUID function 0000\_0005h.

The [rAX] register provides the effective address. The DS segment is the default segment used to create the linear address. Segment overrides may be used with the MONITORX instruction.

The ECX register specifies optional extensions for the MONITORX instruction. There are currently no extensions defined and setting any bits in ECX will result in a #GP exception. The ECX register operand is implicitly 32-bits.

The EDX register specifies optional hints for the MONITORX instruction. There are currently no hints defined and EDX is ignored by the processor. The EDX register operand is implicitly 32-bits.

The MONITORX instruction can be executed at any privilege level and MSR C001\_0015h[MonMwaitUserEn] has no effect on MONITORX.

MONITORX performs the same segmentation and paging checks as a 1-byte read.

Support for the MONITORX instruction is indicated by CPUID Fn0000\_0001\_ECX[MONITORX] = 1.

Software must check the CPUID bit once per program or library initialization before using the MONITORX instruction, or inconsistent behavior may result.

The following pseudo-code shows typical usage of a MONITORX/MWAITX pair:

```
EAX = Linear_Address_to_Monitor;
ECX = 0; // Extensions
EDX = 0; // Hints
while (!matching_store_done){
    MONITORX EAX, ECX, EDX
    IF (!matching_store_done) {
        MWAITX EAX, ECX
    }
}
```

## Setup Monitor Address

Mnemonic	Opcode	Description
MONITORX	0F 01 FA	Establishes a range to be monitored

## Related Instructions

MWAITX, MONITOR, MWAIT

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	MONITORX/MWAITX instructions are not supported, as indicated by CPUID Fn0000_0001(ECX[MONITORX] =0)
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical
	X	X	X	ECX was non-zero
			X	A null data segment was used to reference memory
Page Fault, #PF		X	X	A page fault resulted from the execution of the instruction

**MOV****Move**

Copies an immediate value or the value in a general-purpose register, segment register, or memory location (second operand) to a general-purpose register, segment register, or memory location. The source and destination must be the same size (byte, word, doubleword, or quadword) and cannot both be memory locations.

In opcodes A0 through A3, the memory offsets (called *moffsets*) are address sized. In 64-bit mode, memory offsets default to 64 bits. Opcodes A0–A3, in 64-bit mode, are the only cases that support a 64-bit offset value. (In all other cases, offsets and displacements are a maximum of 32 bits.) The B8 through BF ( $B8 +rq$ ) opcodes, in 64-bit mode, are the only cases that support a 64-bit immediate value (in all other cases, immediate values are a maximum of 32 bits).

When reading segment-registers with a 32-bit operand size, the processor zero-extends the 16-bit selector results to 32 bits. When reading segment-registers with a 64-bit operand size, the processor zero-extends the 16-bit selector to 64 bits. If the destination operand specifies a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector.

It is possible to move a null segment selector value (0000–0003h) into the DS, ES, FS, or GS register. This action does not cause a general protection fault, but a subsequent reference to such a segment *does* cause a #GP exception. For more information about segment selectors, see “Segment Selectors and Registers” in Volume 2.

When the MOV instruction is used to load the SS register, the processor blocks external interrupts until after the execution of the following instruction. This action allows the following instruction to be a MOV instruction to load a stack pointer into the ESP register (MOV ESP, val) before an interrupt occurs. However, the LSS instruction provides a more efficient method of loading SS and ESP.

Attempting to use the MOV instruction to load the CS register generates an invalid opcode exception (#UD). Use the far JMP, CALL, or RET instructions to load the CS register.

To initialize a register to 0, rather than using a MOV instruction, it may be more efficient to use the XOR instruction with identical destination and source operands.

Mnemonic	Opcode	Description
MOV reg/mem8, reg8	88 /r	Move the contents of an 8-bit register to an 8-bit destination register or memory operand.
MOV reg/mem16, reg16	89 /r	Move the contents of a 16-bit register to a 16-bit destination register or memory operand.
MOV reg/mem32, reg32	89 /r	Move the contents of a 32-bit register to a 32-bit destination register or memory operand.
MOV reg/mem64, reg64	89 /r	Move the contents of a 64-bit register to a 64-bit destination register or memory operand.
MOV reg8, reg/mem8	8A /r	Move the contents of an 8-bit register or memory operand to an 8-bit destination register.

Mnemonic	Opcode	Description
MOV <i>reg16, reg/mem16</i>	8B /r	Move the contents of a 16-bit register or memory operand to a 16-bit destination register.
MOV <i>reg32, reg/mem32</i>	8B /r	Move the contents of a 32-bit register or memory operand to a 32-bit destination register.
MOV <i>reg64, reg/mem64</i>	8B /r	Move the contents of a 64-bit register or memory operand to a 64-bit destination register.
MOV <i>reg16/32/64/mem16, segReg</i>	8C /r	Move the contents of a segment register to a 16-bit, 32-bit, or 64-bit destination register or to a 16-bit memory operand.
MOV <i>segReg, reg/mem16</i>	8E /r	Move the contents of a 16-bit register or memory operand to a segment register.
MOV AL, <i>moffset8</i>	A0	Move 8-bit data at a specified memory offset to the AL register.
MOV AX, <i>moffset16</i>	A1	Move 16-bit data at a specified memory offset to the AX register.
MOV EAX, <i>moffset32</i>	A1	Move 32-bit data at a specified memory offset to the EAX register.
MOV RAX, <i>moffset64</i>	A1	Move 64-bit data at a specified memory offset to the RAX register.
MOV <i>moffset8, AL</i>	A2	Move the contents of the AL register to an 8-bit memory offset.
MOV <i>moffset16, AX</i>	A3	Move the contents of the AX register to a 16-bit memory offset.
MOV <i>moffset32, EAX</i>	A3	Move the contents of the EAX register to a 32-bit memory offset.
MOV <i>moffset64, RAX</i>	A3	Move the contents of the RAX register to a 64-bit memory offset.
MOV <i>reg8, imm8</i>	B0 +rb ib	Move an 8-bit immediate value into an 8-bit register.
MOV <i>reg16, imm16</i>	B8 +rw iw	Move a 16-bit immediate value into a 16-bit register.
MOV <i>reg32, imm32</i>	B8 +rd id	Move an 32-bit immediate value into a 32-bit register.
MOV <i>reg64, imm64</i>	B8 +rq iq	Move an 64-bit immediate value into a 64-bit register.
MOV <i>reg/mem8, imm8</i>	C6 /0 ib	Move an 8-bit immediate value to an 8-bit register or memory operand.
MOV <i>reg/mem16, imm16</i>	C7 /0 iw	Move a 16-bit immediate value to a 16-bit register or memory operand.
MOV <i>reg/mem32, imm32</i>	C7 /0 id	Move a 32-bit immediate value to a 32-bit register or memory operand.
MOV <i>reg/mem64, imm32</i>	C7 /0 id	Move a 32-bit signed immediate value to a 64-bit register or memory operand.

## Related Instructions

MOV CR $n$ , MOV DR $n$ , MOVD, MOVSX, MOVZX, MOVSXD, MOVSx

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	An attempt was made to load the CS register.
Segment not present, #NP (selector)			X	The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector, and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	A segment register was loaded, but the segment descriptor exceeded the descriptor table limit.
			X	A segment register was loaded and the segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The SS register was loaded with a null segment selector in non-64-bit mode or while CPL = 3.
			X	The SS register was loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL.
			X	The SS register was loaded and the segment pointed to was not a writable data segment.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was a data or non-conforming code segment, but the RPL or CPL was greater than the DPL.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was not a data segment or readable code segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## MOVBE

## Move Big Endian

Loads or stores a general purpose register while swapping the byte order. Operates on 16-bit, 32-bit, or 64-bit values. Converts big-endian formatted memory data to little-endian format when loading a register and reverses the conversion when storing a GPR to memory.

The load form reads a 16-, 32-, or 64-bit value from memory, swaps the byte order, and places the reordered value in a general-purpose register. When the operand size is 16 bits, the upper word of the destination register remains unchanged. In 64-bit mode, when the operand size is 32 bits, the upper doubleword of the destination register is cleared.

The store form takes a 16-, 32-, or 64-bit value from a general-purpose register, swaps the byte order, and stores the reordered value in the specified memory location. The contents of the source GPR remains unchanged.

In the 16-bit swap, the upper and lower bytes are exchanged. In the doubleword swap operation, bits 7:0 are exchanged with bits 31:24 and bits 15:8 are exchanged with bits 23:16. In the quadword swap operation, bits 7:0 are exchanged with bits 63:56, bits 15:8 with bits 55:48, bits 23:16 with bits 47:40, and bits 31:24 with bits 39:32.

Support for the MOVBE instruction is indicated by CPUID Fn0000\_0001\_ECX[MOVBE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

### Instruction Encoding

Mnemonic	Opcode	Description
MOVBE <i>reg16, mem16</i>	0F 38 F0 /r	Load the low word of a general-purpose register from a 16-bit memory location while swapping the bytes.
MOVBE <i>reg32, mem32</i>	0F 38 F0 /r	Load the low doubleword of a general-purpose register from a 32-bit memory location while swapping the bytes.
MOVBE <i>reg64, mem64</i>	0F 38 F0 /r	Load a 64-bit register from a 64-bit memory location while swapping the bytes.
MOVBE <i>mem16, reg16</i>	0F 38 F1 /r	Store the low word of a general-purpose register to a 16-bit memory location while swapping the bytes.
MOVBE <i>mem32, reg32</i>	0F 38 F1 /r	Store the low doubleword of a general-purpose register to a 32-bit memory location while swapping the bytes.
MOVBE <i>mem64, reg64</i>	0F 38 F1 /r	Store the contents of a 64-bit general-purpose register to a 64-bit memory location while swapping the bytes.

### Related Instruction

BSWAP

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn0000_0001(ECX[MOVBE] = 0).
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## MOVD

## Move Doubleword or Quadword

Moves a 32-bit or 64-bit value in one of the following ways:

- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 or 64 bits of an XMM register, with zero-extension to 128 bits
- from the low-order 32 or 64 bits of an XMM to a 32-bit or 64-bit general-purpose register or memory location
- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 bits (with zero-extension to 64 bits) or the full 64 bits of an MMX register
- from the low-order 32 or the full 64 bits of an MMX register to a 32-bit or 64-bit general-purpose register or memory location

Figure 3-1 on page 230 illustrates the operation of the MOVD instruction.

The MOVD instruction form that moves data to or from MMX registers is part of the MMX instruction subset. Support for MMX instructions is indicated by CPUID Fn0000\_0001\_EDX[MMX] or Fn0000\_0001\_EDX[MMX] = 1.

The MOVD instruction form that moves data to or from XMM registers is part of the SSE2 instruction subset. Support for SSE2 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

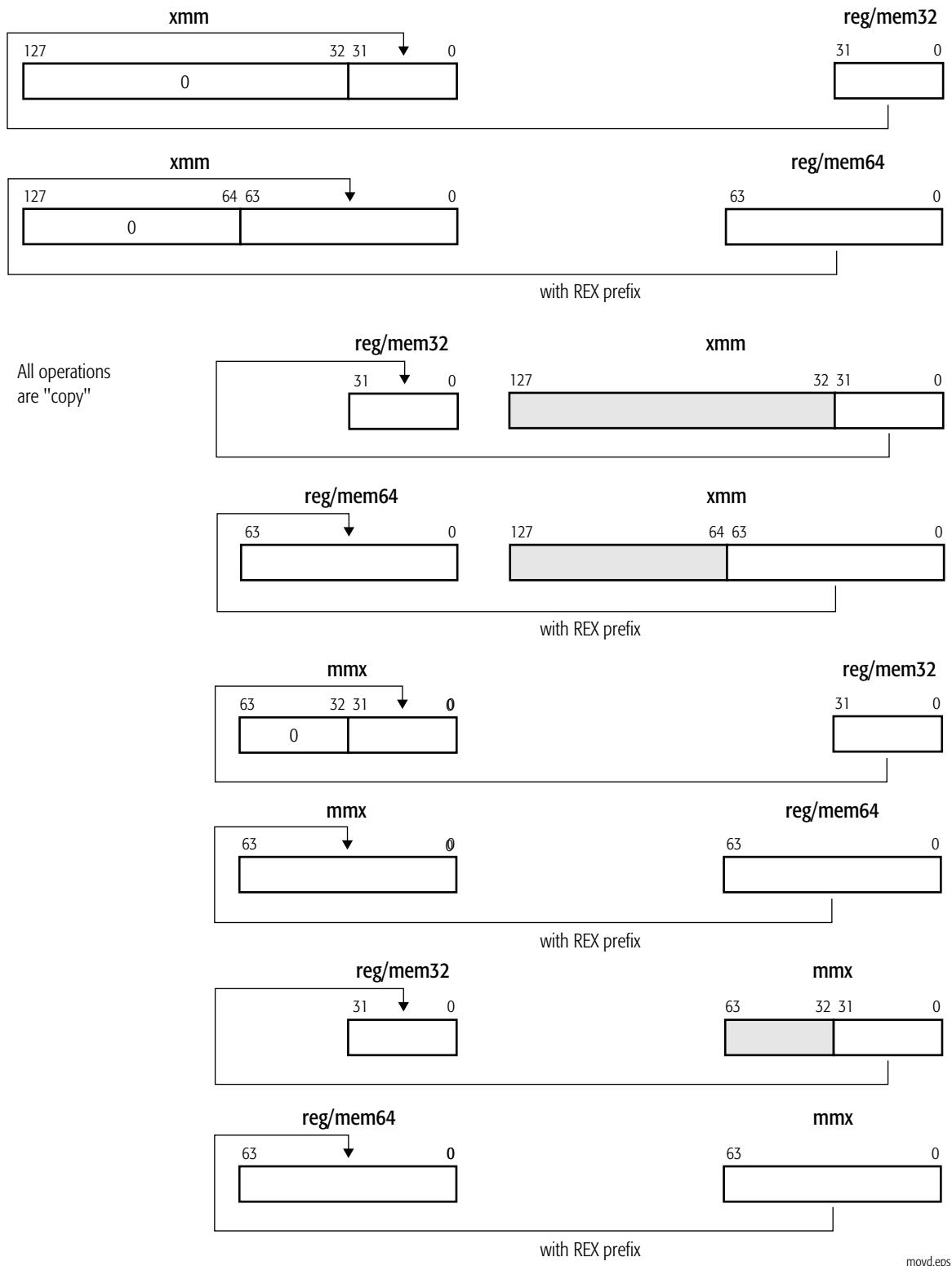


Figure 3-1. MOVD Instruction Operation

## Instruction Encoding

Mnemonic	Opcode	Description
MOVD <i>xmm, reg/mem32</i>	66 0F 6E /r	Move 32-bit value from a general-purpose register or 32-bit memory location to an XMM register.
MOVD <sup>1</sup> <i>xmm, reg/mem64</i>	66 0F 6E /r	Move 64-bit value from a general-purpose register or 64-bit memory location to an XMM register.
MOVD <i>reg/mem32, xmm</i>	66 0F 7E /r	Move 32-bit value from an XMM register to a 32-bit general-purpose register or memory location.
MOVD <sup>1</sup> <i>reg/mem64, xmm</i>	66 0F 7E /r	Move 64-bit value from an XMM register to a 64-bit general-purpose register or memory location.
MOVD <i>mmx, reg/mem32</i>	0F 6E /r	Move 32-bit value from a general-purpose register or 32-bit memory location to an MMX register.
MOVD <i>mmx, reg/mem64</i>	0F 6E /r	Move 64-bit value from a general-purpose register or 64-bit memory location to an MMX register.
MOVD <i>reg/mem32, mmx</i>	0F 7E /r	Move 32-bit value from an MMX register to a 32-bit general-purpose register or memory location.
MOVD <i>reg/mem64, mmx</i>	0F 7E /r	Move 64-bit value from an MMX register to a 64-bit general-purpose register or memory location.

**Note:** 1. Also known as MOVQ in some developer tools.

## Related Instructions

MOVDQA, MOVDQU, MOVDQ2Q, MOVQ, MOVQ2DQ

## rFLAGS Affected

None

## MXCSR Flags Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode, #UD	X	X	X	MMX instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[MMX] or Fn0000_0001_EDX[MMX] = 0.
	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The instruction used XMM registers while CR4.OSFXSR = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

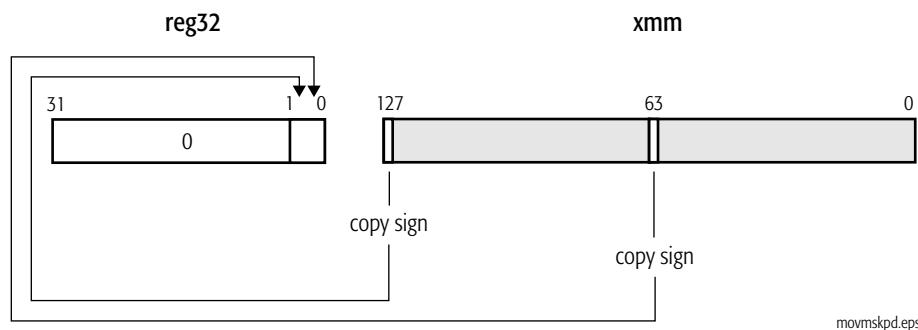
Exception	Real	Virtual 8086	Protected	Description
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An x87 floating-point exception was pending and the instruction referenced an MMX register.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## MOVMSKPD

## Extract Packed Double-Precision Floating-Point Sign Mask

Moves the sign bits of two packed double-precision floating-point values in an XMM register (second operand) to the two low-order bits of a general-purpose register (first operand) with zero-extension.

The function of the MOVMSKPD instruction is illustrated by the diagram below:



The MOVMSKPD instruction is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

### Instruction Encoding

Mnemonic	Opcode	Description
MOVMSKPD <i>reg32, xmm</i>	66 0F 50 /r	Move sign bits 127 and 63 in an XMM register to a 32-bit general-purpose register.

### Related Instructions

MOVMSKPS, PMOVMSKB

### rFLAGS Affected

None

### MXCSR Flags Affected

None

## Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

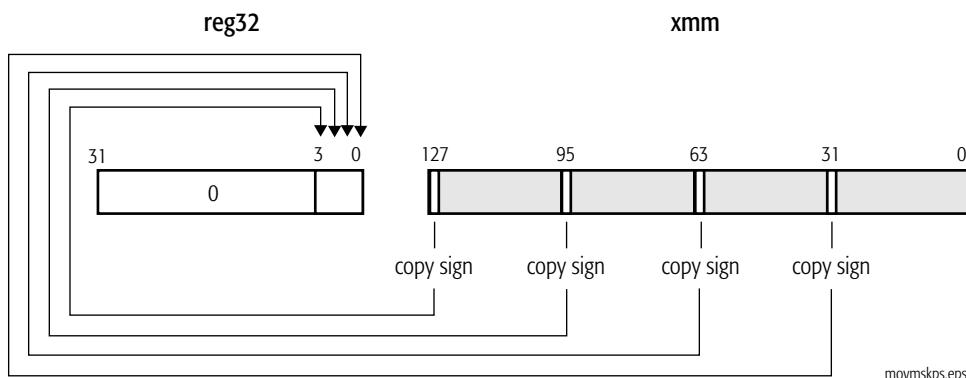
**MOVMSKPS****Extract Packed Single-Precision Floating-Point Sign Mask**

Moves the sign bits of four packed single-precision floating-point values in an XMM register (second operand) to the four low-order bits of a general-purpose register (first operand) with zero-extension.

The MOVMSKPD instruction is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
MOVMSKPS <i>reg32, xmm</i>	0F 50 /r	Move sign bits 127, 95, 63, 31 in an XMM register to a 32-bit general-purpose register.

**Related Instructions**

MOVMSKPD, PMOVMSKB

**rFLAGS Affected**

None

**MXCSR Flags Affected**

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

**MOVNTI****Move Non-Temporal Doubleword or Quadword**

Stores a value in a 32-bit or 64-bit general-purpose register (second operand) in a memory location (first operand). This instruction indicates to the processor that the data is non-temporal and is unlikely to be used again soon. The processor treats the store as a write-combining (WC) memory write, which minimizes cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The MOVNTI instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE instruction to force strong memory ordering of MOVNTI with respect to other stores.

The MOVNTI instruction is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
MOVNTI <i>mem32, reg32</i>	0F C3 /r	Stores a 32-bit general-purpose register value into a 32-bit memory location, minimizing cache pollution.
MOVNTI <i>mem64, reg64</i>	0F C3 /r	Stores a 64-bit general-purpose register value into a 64-bit memory location, minimizing cache pollution.

**Related Instructions**

MOVNTDQ, MOVNTPD, MOVNTPS, MOVNTQ

**rFLAGS Affected**

None

**Exceptions**

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## **MOVS MOVSB MOVSW MOVSD MOVSQ**

## **Move String**

Moves a byte, word, doubleword, or quadword from the memory location pointed to by DS:rSI to the memory location pointed to by ES:rDI, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments both pointers; otherwise, it decrements them. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the MOVSx instruction with explicit operands address the first operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but can be overridden by a segment prefix. These instructions always address the second operand at ES:[rDI] (ES may not be overridden). The explicit operands serve only to specify the type (size) of the value being moved.

The no-operands forms of the instruction use the DS:[rSI] and ES:[rDI] registers to point to the value to be moved (they do not allow a segment prefix). The mnemonic determines the size of the operands.

Do not confuse this MOVSD instruction with the same-mnemonic MOVSD (move scalar double-precision floating-point) instruction in the 128-bit media instruction set. Assemblers can distinguish the instructions by the number and type of operands.

The MOVSx instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 12.

<b>Mnemonic</b>	<b>Opcode</b>	<b>Description</b>
MOVS <i>mem8, mem8</i>	A4	Move byte at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVS <i>mem16, mem16</i>	A5	Move word at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVS <i>mem32, mem32</i>	A5	Move doubleword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVS <i>mem64, mem64</i>	A5	Move quadword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVSB	A4	Move byte at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVSW	A5	Move word at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.

Mnemonic	Opcode	Description
MOVSD	A5	Move doubleword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVSQ	A5	Move quadword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.

## Related Instructions

MOV, LODSx, STOSx

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## MOVsx

## Move with Sign-Extension

Copies the value in a register or memory location (second operand) into a register (first operand), extending the most significant bit of an 8-bit or 16-bit value into all higher bits in a 16-bit, 32-bit, or 64-bit register.

Mnemonic	Opcode	Description
MOVsx reg16, reg/mem8	0F BE /r	Move the contents of an 8-bit register or memory location to a 16-bit register with sign extension.
MOVsx reg32, reg/mem8	0F BE /r	Move the contents of an 8-bit register or memory location to a 32-bit register with sign extension.
MOVsx reg64, reg/mem8	0F BE /r	Move the contents of an 8-bit register or memory location to a 64-bit register with sign extension.
MOVsx reg32, reg/mem16	0F BF /r	Move the contents of an 16-bit register or memory location to a 32-bit register with sign extension.
MOVsx reg64, reg/mem16	0F BF /r	Move the contents of an 16-bit register or memory location to a 64-bit register with sign extension.

### Related Instructions

MOVSXD, MOVZX

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## MOVSXD

## Move with Sign-Extend Doubleword

Copies the 32-bit value in a register or memory location (second operand) into a 64-bit register (first operand), extending the most significant bit of the 32-bit value into all higher bits of the 64-bit register.

This instruction requires the REX prefix 64-bit operand size bit (REX.W) to be set to 1 to sign-extend a 32-bit source operand to a 64-bit result. Without the REX operand-size prefix, the operand size will be 32 bits, the default for 64-bit mode, and the source is zero-extended into a 64-bit register. With a 16-bit operand size, only 16 bits are copied, without modifying the upper 48 bits in the destination.

This instruction is available only in 64-bit mode. In legacy or compatibility mode this opcode is interpreted as ARPL.

Mnemonic	Opcode	Description
MOVSXD reg64, reg/mem32	63 /r	Move the contents of a 32-bit register or memory operand to a 64-bit register with sign extension.

### Related Instructions

MOVSX, MOVZX

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS			X	A memory address was non-canonical.
General protection, #GP			X	A memory address was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## MOVZX

## Move with Zero-Extension

Copies the value in a register or memory location (second operand) into a register (first operand), zero-extending the value to fit in the destination register. The operand-size attribute determines the size of the zero-extended value.

Mnemonic	Opcode	Description
MOVZX reg16, reg/mem8	0F B6 /r	Move the contents of an 8-bit register or memory operand to a 16-bit register with zero-extension.
MOVZX reg32, reg/mem8	0F B6 /r	Move the contents of an 8-bit register or memory operand to a 32-bit register with zero-extension.
MOVZX reg64, reg/mem8	0F B6 /r	Move the contents of an 8-bit register or memory operand to a 64-bit register with zero-extension.
MOVZX reg32, reg/mem16	0F B7 /r	Move the contents of a 16-bit register or memory operand to a 32-bit register with zero-extension.
MOVZX reg64, reg/mem16	0F B7 /r	Move the contents of a 16-bit register or memory operand to a 64-bit register with zero-extension.

### Related Instructions

MOVSXD, MOVSX

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**MUL****Unsigned Multiply**

Multiplies the unsigned byte, word, doubleword, or quadword value in the specified register or memory location by the value in AL, AX, EAX, or RAX and stores the result in AX, DX:AX, EDX:EAX, or RDX:RAX (depending on the operand size). It puts the high-order bits of the product in AH, DX, EDX, or RDX.

If the upper half of the product is non-zero, the instruction sets the carry flag (CF) and overflow flag (OF) both to 1. Otherwise, it clears CF and OF to 0. The other arithmetic flags (SF, ZF, AF, PF) are undefined.

Mnemonic	Opcode	Description
MUL reg/mem8	F6 /4	Multiplies an 8-bit register or memory operand by the contents of the AL register and stores the result in the AX register.
MUL reg/mem16	F7 /4	Multiplies a 16-bit register or memory operand by the contents of the AX register and stores the result in the DX:AX register.
MUL reg/mem32	F7 /4	Multiplies a 32-bit register or memory operand by the contents of the EAX register and stores the result in the EDX:EAX register.
MUL reg/mem64	F7 /4	Multiplies a 64-bit register or memory operand by the contents of the RAX register and stores the result in the RDX:RAX register.

**Related Instructions**

DIV

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference is performed while alignment checking was enabled.

## MULX

## Multiply Unsigned

Computes the unsigned product of the specified source operand and the implicit source operand rDX. Writes the upper half of the product to the first destination and the lower half to the second. Does not affect the arithmetic flags.

This instruction has three operands:

**MULX dest1, dest2, src**

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The first and second operands (*dest1* and *dest2*) are general purpose registers. The specified source operand (*src*) is either a general purpose register or a memory operand. If the first and second operands specify the same register, the register receives the upper half of the product.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
MULX reg32, reg32, reg/mem32	C4	$\overline{\text{RXB}}.02$	0. $\overline{\text{dest2}}$ .0.11	F6 /r
MULX reg64, reg64, reg/mem64	C4	$\overline{\text{RXB}}.02$	1. $\overline{\text{dest2}}$ .0.11	F6 /r

## Related Instructions

### rFLAGS Affected

None.

### Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## MWAITX

## Monitor Wait with Timeout

Used in conjunction with the MONITORX instruction to cause a processor to wait until a store occurs to a specific linear address range from another processor or the timer expires. The previously executed MONITORX instruction causes the processor to enter the monitor event pending state. The MWAITX instruction may enter an implementation dependent power state until the monitor event pending state is exited. The MWAITX instruction has the same effect on architectural state as the NOP instruction.

Events that cause an exit from the monitor event pending state include:

- A store from another processor matches the address range established by the MONITORX instruction.
- The timer expires.
- Any unmasked interrupt, including INTR, NMI, SMI, INIT.
- RESET.
- Any far control transfer that occurs between the MONITORX and the MWAITX.

EAX specifies optional hints for the MWAITX instruction. Optimized C-state request is communicated through EAX[7:4]. The processor C-state is EAX[7:4]+1, so to request C0 is to place the value F in EAX[7:4] and to request C1 is to place the value 0 in EAX[7:4]. All other components of EAX should be zero when making the C1 request. Setting a reserved bit in EAX is ignored by the processor.

ECX specifies optional extensions for the MWAITX instruction. The extensions currently defined for ECX are:

- Bit 0: When set, allows interrupts to wake MWAITX, even when eFLAGS.IF = 0. Support for this extension is indicated by a feature flag returned by the CPUID instruction.
- Bit 1: When set, EBX contains the maximum wait time expressed in Software P0 clocks, the same clocks counted by the TSC. Setting bit 1 but passing in a value of zero on EBX is equivalent to setting bit 1 to a zero. The timer will not be an exit condition.
- Bit 31-2: When non-zero, results in a #GP(0) exception.

CPUID Function 0000\_0005h indicates support for extended features of MONITORX/MWAITX as well as MONITOR/MWAIT:

- CPUID Fn0000\_0005\_ECX[EMX] = 1 indicates support for enumeration of MONITOR/MWAIT/MONITORX/MWAITX extensions.
- CPUID Fn0000\_0005\_ECX[IBE] = 1 indicates that MWAIT/MWAITX can set ECX[0] to allow interrupts to cause an exit from the monitor event pending state even when eFLAGS.IF = 0.

The MWAITX instruction can be executed at any privilege level and MSR C001\_0015h[MonMwaitUserEn] has no effect on MWAITX.

Support for the MWAITX instruction is indicated by CPUID Fn0000\_0001\_ECX[MONITORX] = 1.

Software must check the CPUID bit once per program or library initialization before using the MWAITX instruction, or inconsistent behavior may result.

The use of the MWAITX instruction is contingent upon the satisfaction of the following coding requirements:

- MONITORX must precede the MWAITX and occur in the same loop.
- MWAITX must be conditionally executed only if the awaited store has not already occurred. (This prevents a race condition between the MONITORX instruction arming the monitoring hardware and the store intended to trigger the monitoring hardware.)

There is no indication after exiting MWAITX of why the processor exited or if the timer expired. It is up to software to check whether the awaiting store has occurred, and if not, determining how much time has elapsed if it wants to re-establish the MONITORX with a new timer value.

Mnemonic	Opcode	Description
MWAITX	0F 01 FB	Causes the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events

## Related Instructions

MONITORX, MONITOR, MWAIT

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	MONITORX/MWAITX instructions are not supported, as indicated by CPUID Fn0000_0001(ECX[MONITORX]=0)
General protection, #GP	X	X	X	Unsupported extension bits in ECX

**NEG****Two's Complement Negation**

Performs the two's complement negation of the value in the specified register or memory location by subtracting the value from 0. Use this instruction only on signed integer numbers.

If the value is 0, the instruction clears the CF flag to 0; otherwise, it sets CF to 1. The OF, SF, ZF, AF, and PF flag settings depend on the result of the operation.

The forms of the NEG instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
NEG reg/mem8	F6 /3	Performs a two's complement negation on an 8-bit register or memory operand.
NEG reg/mem16	F7 /3	Performs a two's complement negation on a 16-bit register or memory operand.
NEG reg/mem32	F7 /3	Performs a two's complement negation on a 32-bit register or memory operand.
NEG reg/mem64	F7 /3	Performs a two's complement negation on a 64-bit register or memory operand.

**Related Instructions**

AND, NOT, OR, XOR

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand is in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## NOP

## No Operation

Does nothing. This instruction increments the rIP to point to next instruction, but does not affect the machine state in any other way.

The single-byte variant is an alias for XCHG rAX, rAX.

Mnemonic	Opcode	Description
NOP	90	Performs no operation.
NOP reg/mem16	0F 1F /0	Performs no operation on a 16-bit register or memory operand.
NOP reg/mem32	0F 1F /0	Performs no operation on a 32-bit register or memory operand.
NOP reg/mem64	0F 1F /0	Performs no operation on a 64-bit register or memory operand.

## Related Instructions

None

## rFLAGS Affected

None

## Exceptions

None

**NOT****One's Complement Negation**

Performs the one's complement negation of the value in the specified register or memory location by inverting each bit of the value.

The memory-operand forms of the NOT instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
NOT reg/mem8	F6 /2	Complements the bits in an 8-bit register or memory operand.
NOT reg/mem16	F7 /2	Complements the bits in a 16-bit register or memory operand.
NOT reg/mem32	F7 /2	Complements the bits in a 32-bit register or memory operand.
NOT reg/mem64	F7 /2	Compliments the bits in a 64-bit register or memory operand.

**Related Instructions**

AND, NEG, OR, XOR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference is performed while alignment checking was enabled.

**OR****Logical OR**

Performs a logical **or** on the bits in a register, memory location, or immediate value (second operand) and a register or memory location (first operand) and stores the result in the first operand location. The two operands cannot both be memory locations.

If both corresponding bits are 0, the corresponding bit of the result is 0; otherwise, the corresponding result bit is 1.

The forms of the OR instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
OR AL, <i>imm8</i>	0C <i>ib</i>	<b>or</b> the contents of AL with an immediate 8-bit value.
OR AX, <i>imm16</i>	0D <i>iw</i>	<b>or</b> the contents of AX with an immediate 16-bit value.
OR EAX, <i>imm32</i>	0D <i>id</i>	<b>or</b> the contents of EAX with an immediate 32-bit value.
OR RAX, <i>imm32</i>	0D <i>id</i>	<b>or</b> the contents of RAX with a sign-extended immediate 32-bit value.
OR <i>reg/mem8, imm8</i>	80 /1 <i>ib</i>	<b>or</b> the contents of an 8-bit register or memory operand and an immediate 8-bit value.
OR <i>reg/mem16, imm16</i>	81 /1 <i>iw</i>	<b>or</b> the contents of a 16-bit register or memory operand and an immediate 16-bit value.
OR <i>reg/mem32, imm32</i>	81 /1 <i>id</i>	<b>or</b> the contents of a 32-bit register or memory operand and an immediate 32-bit value.
OR <i>reg/mem64, imm32</i>	81 /1 <i>id</i>	<b>or</b> the contents of a 64-bit register or memory operand and sign-extended immediate 32-bit value.
OR <i>reg/mem16, imm8</i>	83 /1 <i>ib</i>	<b>or</b> the contents of a 16-bit register or memory operand and a sign-extended immediate 8-bit value.
OR <i>reg/mem32, imm8</i>	83 /1 <i>ib</i>	<b>or</b> the contents of a 32-bit register or memory operand and a sign-extended immediate 8-bit value.
OR <i>reg/mem64, imm8</i>	83 /1 <i>ib</i>	<b>or</b> the contents of a 64-bit register or memory operand and a sign-extended immediate 8-bit value.
OR <i>reg/mem8, reg8</i>	08 /r	<b>or</b> the contents of an 8-bit register or memory operand with the contents of an 8-bit register.
OR <i>reg/mem16, reg16</i>	09 /r	<b>or</b> the contents of a 16-bit register or memory operand with the contents of a 16-bit register.
OR <i>reg/mem32, reg32</i>	09 /r	<b>or</b> the contents of a 32-bit register or memory operand with the contents of a 32-bit register.
OR <i>reg/mem64, reg64</i>	09 /r	<b>or</b> the contents of a 64-bit register or memory operand with the contents of a 64-bit register.

Mnemonic	Opcode	Description
OR reg8, reg/mem8	0A /r	or the contents of an 8-bit register with the contents of an 8-bit register or memory operand.
OR reg16, reg/mem16	0B /r	or the contents of a 16-bit register with the contents of a 16-bit register or memory operand.
OR reg32, reg/mem32	0B /r	or the contents of a 32-bit register with the contents of a 32-bit register or memory operand.
OR reg64, reg/mem64	0B /r	or the contents of a 64-bit register with the contents of a 64-bit register or memory operand.

The following chart summarizes the effect of this instruction:

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

## Related Instructions

AND, NEG, NOT, XOR

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## OUT

## Output to Port

Copies the value from the AL, AX, or EAX register (second operand) to an I/O port (first operand). The port address can be a byte-immediate value (00h to FFh) or the value in the DX register (0000h to FFFFh). The source register used determines the size of the port (8, 16, or 32 bits).

If the operand size is 64 bits, OUT only writes to a 32-bit I/O port.

If the CPL is higher than the IOPL or the mode is virtual mode, OUT checks the I/O permission bitmap in the TSS before allowing access to the I/O port. See Volume 2 for details on the TSS I/O permission bitmap.

Mnemonic	Opcode	Description
OUT <i>imm8</i> , AL	E6 <i>ib</i>	Output the byte in the AL register to the port specified by an 8-bit immediate value.
OUT <i>imm8</i> , AX	E7 <i>ib</i>	Output the word in the AX register to the port specified by an 8-bit immediate value.
OUT <i>imm8</i> , EAX	E7 <i>ib</i>	Output the doubleword in the EAX register to the port specified by an 8-bit immediate value.
OUT DX, AL	EE	Output byte in AL to the output port specified in DX.
OUT DX, AX	EF	Output word in AX to the output port specified in DX.
OUT DX, EAX	EF	Output doubleword in EAX to the output port specified in DX.

## Related Instructions

IN, INS<sub>x</sub>, OUTS<sub>x</sub>

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		One or more I/O permission bits were set in the TSS for the accessed port.
			X	The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port.
Page fault (#PF)		X	X	A page fault resulted from the execution of the instruction.

## **OUTS OUTSB OUTSW OUTSD**

## **Output String**

Copies data from the memory location pointed to by DS:rSI to the I/O port address (0000h to FFFFh) specified in the DX register, and then increments or decrements the rSI register according to the setting of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements the pointer by 1, 2, or 4, depending on the size of the value being copied.

The OUTS DX mnemonic uses an explicit memory operand (second operand) to determine the type (size) of the value being copied, but always uses DS:rSI for the location of the value to copy. The explicit register operand (first operand) specifies the I/O port address and must always be DX.

The no-operands forms of the mnemonic use the DS:rSI register pair to point to the memory data to be copied and the contents of the DX register as the destination I/O port address. The mnemonic specifies the size of the I/O port and the type (size) of the value being copied.

The OUTSx instruction supports the REP prefix. For details about the REP prefix, see “Repeat Prefixes” on page 12.

If the effective operand size is 64-bits, the instruction behaves as if the operand size were 32 bits.

If the CPL is higher than the IOPL or the mode is virtual mode, OUTSx checks the I/O permission bitmap in the TSS before allowing access to the I/O port. See Volume 2 for details on the TSS I/O permission bitmap.

Mnemonic	Opcode	Description
OUTS DX, <i>mem8</i>	6E	Output the byte in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTS DX, <i>mem16</i>	6F	Output the word in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTS DX, <i>mem32</i>	6F	Output the doubleword in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTSB	6E	Output the byte in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTSW	6F	Output the word in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTSD	6F	Output the doubleword in DS:rSI to the port specified in DX, then increment or decrement rSI.

**Related Instructions**IN, INS $x$ , OUT**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
		X		One or more I/O permission bits were set in the TSS for the accessed port.
			X	The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference is performed while alignment checking was enabled.

## PAUSE

## Pause

Improves the performance of spin loops, by providing a hint to the processor that the current code is in a spin loop. The processor may use this to optimize power consumption while in the spin loop.

Architecturally, this instruction behaves like a NOP instruction.

Processors that do not support PAUSE treat this opcode as a NOP instruction.

Mnemonic	Opcode	Description
PAUSE	F3 90	Provides a hint to processor that a spin loop is being executed.

### Related Instructions

None

### rFLAGS Affected

None

### Exceptions

None

## PDEP

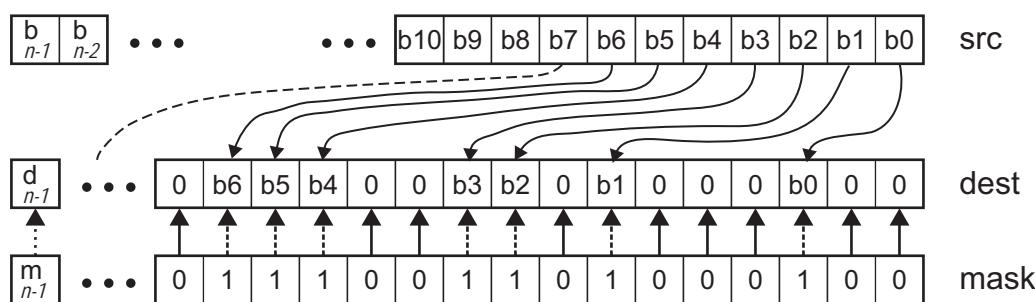
## Parallel Deposit Bits

Scatters consecutive bits of the first source operand, starting at the least significant bit, to bit positions in the destination as specified by 1 bits in the second source operand (*mask*). Bit positions in the destination corresponding to 0 bits in the mask are cleared.

This instruction has three operands:

**PDEP dest, src, mask**

The following diagram illustrates the operation of this instruction.



v3\_PDEP\_instruct.eps

If the mask is all ones, the execution of this instruction effectively copies the source to the destination.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) and the source (*src*) are general-purpose registers. The second source operand (*mask*) is either a general-purpose register or a memory operand.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
PDEP reg32, reg32, reg/mem32	C4	$\overline{\text{RXB}}.02$	0. $\overline{\text{src}}$ .0.11	F5 /r
PDEP reg64, reg64, reg/mem64	C4	$\overline{\text{RXB}}.02$	1. $\overline{\text{src}}$ .0.11	F5 /r

## Related Instructions

### rFLAGS Affected

None.

### Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## PEXT

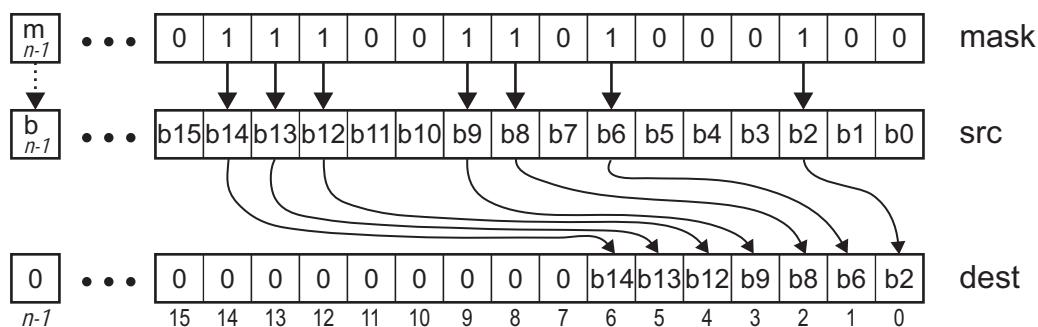
## Parallel Extract Bits

Copies bits from the source operand, based on a mask, and packs them into the low-order bits of the destination. Clears all bits in the destination to the left of the most-significant bit copied.

This instruction has three operands:

**PEXT dest, src, mask**

The following diagram illustrates the operation of this instruction.



v3\_PEXT\_instruct.eps

If the mask is all ones, the execution of this instruction effectively copies the source to the destination.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) and the source (*src*) are general-purpose registers. The second source operand (*mask*) is either a general-purpose register or a memory operand.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
PEXT reg32, reg32, reg/mem32	C4	$\overline{\text{RXB}}.02$	0. $\overline{\text{src}}$ .0.10	F5 /r
PEXT reg64, reg64, reg/mem64	C4	$\overline{\text{RXB}}.02$	1. $\overline{\text{src}}$ .0.10	F5 /r

## Related Instructions

### rFLAGS Affected

None.

### Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**POP****Pop Stack**

Copies the value pointed to by the stack pointer (SS:rSP) to the specified register or memory location and then increments the rSP by 2 for a 16-bit pop, 4 for a 32-bit pop, or 8 for a 64-bit pop.

The operand-size attribute determines the amount by which the stack pointer is incremented (2, 4 or 8 bytes). The stack-size attribute determines whether SP, ESP, or RSP is incremented.

For forms of the instruction that load a segment register (POP DS, POP ES, POP FS, POP GS, POP SS), the source operand must be a valid segment selector. When a segment selector is popped into a segment register, the processor also loads all associated descriptor information into the hidden part of the register and validates it.

It is possible to pop a null segment selector value (0000–0003h) into the DS, ES, FS, or GS register. This action does not cause a general protection fault, but a subsequent reference to such a segment *does* cause a #GP exception. For more information about segment selectors, see "Segment Selectors and Registers" in *Volume 2: System Programming*.

In 64-bit mode, the POP operand size defaults to 64 bits and there is no prefix available to encode a 32-bit operand size. Using POP DS, POP ES, or POP SS instruction in 64-bit mode generates an invalid-opcode exception.

This instruction cannot pop a value into the CS register. The RET (Far) instruction performs this function.

Mnemonic	Opcode	Description
POP reg/mem16	8F /0	Pop the top of the stack into a 16-bit register or memory location.
POP reg/mem32	8F /0	Pop the top of the stack into a 32-bit register or memory location. (No prefix for encoding this in 64-bit mode.)
POP reg/mem64	8F /0	Pop the top of the stack into a 64-bit register or memory location.
POP reg16	58 +rw	Pop the top of the stack into a 16-bit register.
POP reg32	58 +rd	Pop the top of the stack into a 32-bit register. (No prefix for encoding this in 64-bit mode.)
POP reg64	58 +rq	Pop the top of the stack into a 64-bit register.
POP DS	1F	Pop the top of the stack into the DS register. (Invalid in 64-bit mode.)
POP ES	07	Pop the top of the stack into the ES register. (Invalid in 64-bit mode.)
POP SS	17	Pop the top of the stack into the SS register. (Invalid in 64-bit mode.)

Mnemonic	Opcode	Description
POP FS	0F A1	Pop the top of the stack into the FS register.
POP GS	0F A9	Pop the top of the stack into the GS register.

## Related Instructions

PUSH

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	POP DS, POP ES, or POP SS was executed in 64-bit mode.
Segment not present, #NP (selector)			X	The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	A segment register was loaded and the segment descriptor exceeded the descriptor table limit.
			X	A segment register was loaded and the segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The SS register was loaded with a null segment selector in non-64-bit mode or while CPL = 3.
			X	The SS register was loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL.
			X	The SS register was loaded and the segment pointed to was not a writable data segment.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was a data or non-conforming code segment, but the RPL or the CPL was greater than the DPL.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was not a data segment or readable code segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## POPA POPAD

## POP All GPRs

Pops words or doublewords from the stack into the general-purpose registers in the following order: eDI, eSI, eBP, eSP (image is popped and discarded), eBX, eDX, eCX, and eAX. The instruction increments the stack pointer by 16 or 32, depending on the operand size.

Using the POPA or POPAD instructions in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
POPA	61	Pop the DI, SI, BP, SP, BX, DX, CX, and AX registers. (Invalid in 64-bit mode.)
POPAD	61	Pop the EDI, ESI, EBP, ESP, EBX, EDX, ECX, and EAX registers. (Invalid in 64-bit mode.)

### Related Instructions

PUSHA, PUSHAD

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode (#UD)			X	This instruction was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## POPCNT

## Bit Population Count

Counts the number of bits having a value of 1 in the source operand and places the result in the destination register. The source operand is a 16-, 32-, or 64-bit general purpose register or memory operand; the destination operand is a general purpose register of the same size as the source operand register.

If the input operand is zero, the ZF flag is set to 1 and zero is written to the destination register. Otherwise, the ZF flag is cleared. The other flags are cleared.

Support for the POPCNT instruction is indicated by CPUID Fn0000\_0001\_ECX[POPCNT] = 1. Software MUST check the CPUID bit once per program or library initialization before using the POPCNT instruction, or inconsistent behavior may result.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic		Opcode	Description
POPCNT	<i>reg16, reg/mem16</i>	F3 OF B8 /r	Count the 1s in reg/mem16.
POPCNT	<i>reg32, reg/mem32</i>	F3 OF B8 /r	Count the 1s in reg/mem32.
POPCNT	<i>reg64, reg/mem64</i>	F3 OF B8 /r	Count the 1s in reg/mem64.

### Related Instructions

BSF, BSR, LZCNT

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	M	0	0	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The POPCNT instruction is not supported, as indicated by CPUID Fn0000_0001_ECX[POPCNT].
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**POPF****POPFD****POPFQ****POP to rFLAGS**

Pops a word, doubleword, or quadword from the stack into the rFLAGS register and then increments the stack pointer by 2, 4, or 8, depending on the operand size.

In protected or real mode, all the non-reserved flags in the rFLAGS register can be modified, except the VIP, VIF, and VM flags, which are unchanged. In protected mode, at a privilege level greater than 0 the IOPL is also unchanged. The instruction alters the interrupt flag (IF) only when the CPL is less than or equal to the IOPL.

In virtual-8086 mode, if IOPL field is less than 3, attempting to execute a POPFx or PUSHFx instruction while VME is not enabled, or the operand size is not 16-bit, generates a #GP exception.

In 64-bit mode, this instruction defaults to a 64-bit operand size; there is no prefix available to encode a 32-bit operand size.

Mnemonic	Opcode	Description
POPF	9D	Pop a word from the stack into the FLAGS register.
POPFD	9D	Pop a double word from the stack into the EFLAGS register. (No prefix for encoding this in 64-bit mode.)
POPFQ	9D	Pop a quadword from the stack to the RFLAGS register.

**Action**

```
// See "Pseudocode Definition" on page 57.

POPF_START:

IF (REAL_MODE)
    POPF_REAL
ELSIF (PROTECTED_MODE)
    POPF_PROTECTED
ELSE // (VIRTUAL_MODE)
    POPF_VIRTUAL

POPF_REAL:

POP.v temp_RFLAGS
RFLAGS.v = temp_RFLAGS          // VIF,VIP,VM unchanged
                                // RF cleared
EXIT
```

POPF\_PROTECTED:

```
POP.v temp_RFLAGS
RFLAGS.v = temp_RFLAGS          // VIF,VIP,VM unchanged
                                // IOPL changed only if (CPL==0)
                                // IF changed only if (CPL<=old_RFLAGS.IOPL)
                                // RF cleared
EXIT
```

POPF\_VIRTUAL:

```
IF (RFLAGS.IOPL==3)
{
    POP.v temp_RFLAGS
    RFLAGS.v = temp_RFLAGS          // VIF,VIP,VM,IOPL unchanged
                                    // RF cleared
    EXIT
}
ELSIF ((CR4.VME==1) && (OPERAND_SIZE==16))
{
    POP.w temp_RFLAGS
    IF (((temp_RFLAGS.IF==1) && (RFLAGS.VIP==1)) || (temp_RFLAGS.TF==1))
        EXCEPTION [#GP(0)]           // notify the virtual-mode-manager to
deliver
                                            // the task's pending interrupts
    RFLAGS.w = temp_RFLAGS          // IF,IOPL unchanged
                                    // RFLAGS.VIF=temp_RFLAGS.IF
                                    // RF cleared
    EXIT
}
ELSE // ((RFLAGS.IOPL<3) && ((CR4.VME==0) || (OPERAND_SIZE!=16)))
    EXCEPTION [#GP(0)]
```

## Related Instructions

PUSHF, PUSHFD, PUSHFQ

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M		M	M		0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP		X		The I/O privilege level was less than 3 and one of the following conditions was true: <ul style="list-style-type: none"> <li>• CR4.VME was 0.</li> <li>• The effective operand size was 32-bit.</li> <li>• Both the original EFLAGS.VIP and the new EFLAGS.IF bits were set.</li> <li>• The new EFLAGS.TF bit was set.</li> </ul>
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## PREFETCH PREFETCHW

Loads the entire 64-byte aligned memory sequence *containing* the specified memory address into the L1 data cache. The position of the specified memory address within the 64-byte cache line is irrelevant. If a cache hit occurs, or if a memory fault is detected, no bus cycle is initiated and the instruction is treated as a NOP.

The PREFETCHW instruction loads the prefetched line and sets the cache-line state to Modified, in anticipation of subsequent data writes to the line. The PREFETCH instruction, by contrast, typically sets the cache-line state to Exclusive (depending on the hardware implementation).

The opcodes for the PREFETCH/PREFETCHW instructions include the ModRM byte; however, only the memory form of ModRM is valid. The register form of ModRM causes an invalid-opcode exception. Because there is no destination register, the three destination register field bits of the ModRM byte define the type of prefetch to be performed. The bit patterns 000b and 001b define the PREFETCH and PREFETCHW instructions, respectively. All other bit patterns are reserved for future use.

The *reserved* PREFETCH types do not result in an invalid-opcode exception if executed. Instead, for forward compatibility with future processors that may implement additional forms of the PREFETCH instruction, all reserved PREFETCH types are implemented as synonyms of the basic PREFETCH type (the PREFETCH instruction with type 000b).

The operation of these instructions is implementation-dependent. The processor implementation can ignore or change these instructions. The size of the cache line also depends on the implementation, with a minimum size of 32 bytes. For details on the use of this instruction, see the processor data sheets or other software-optimization documentation relating to particular hardware implementations.

When paging is enabled and PREFETCHW performs a prefetch from a writable page, it may set the PTE Dirty bit to 1.

Support for the PREFETCH and PREFETCHW instructions is indicated by CPUID Fn8000\_0001\_ECX[3DNowPrefetch] OR Fn8000\_0001\_EDX[LM] OR Fn8000\_0001\_EDX[3DNow] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

## Prefetch L1 Data-Cache Line

Mnemonic	Opcode	Description
PREFETCH mem8	0F 0D /0	Prefetch processor cache line into L1 data cache.
PREFETCHW mem8	0F 0D /1	Prefetch processor cache line into L1 data cache and mark it modified.

## Related Instructions

PREFETCH*level*

## rFLAGS Affected

None

## Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	PREFETCH and PREFETCHW instructions are not supported, as indicated by CPUID Fn8000_0001(ECX[3DNowPrefetch]) AND Fn8000_0001(EDX[LM]) AND Fn8000_0001(EDX[3DNow]) = 0.
	X	X	X	The operand was a register.

**PREFETCH/level****Prefetch Data to Cache Level /level**

Loads a cache line from the specified memory address into the data-cache level specified by the locality reference bits 5:3 of the ModRM byte. Table 3-3 on page 275 lists the locality reference options for the instruction.

This instruction loads a cache line even if the *mem8* address is not aligned with the start of the line. If the cache line is already contained in a cache level that is lower than the specified locality reference, or if a memory fault is detected, a bus cycle is not initiated and the instruction is treated as a NOP.

The operation of this instruction is implementation-dependent. The processor implementation can ignore or change this instruction. The size of the cache line also depends on the implementation, with a minimum size of 32 bytes. AMD processors alias PREFETCH1 and PREFETCH2 to PREFETCH0. For details on the use of this instruction, see the software-optimization documentation relating to particular hardware implementations.

Mnemonic	Opcode	Description
PREFETCHNTA <i>mem8</i>	0F 18 /0	Move data closer to the processor using the NTA reference.
PREFETCHT0 <i>mem8</i>	0F 18 /1	Move data closer to the processor using the T0 reference.
PREFETCHT1 <i>mem8</i>	0F 18 /2	Move data closer to the processor using the T1 reference.
PREFETCHT2 <i>mem8</i>	0F 18 /3	Move data closer to the processor using the T2 reference.

**Table 3-3. Locality References for the Prefetch Instructions**

Locality Reference	Description
NTA	Non-Temporal Access—Move the specified data into the processor with minimum cache pollution. This is intended for data that will be used only once, rather than repeatedly. The specific technique for minimizing cache pollution is implementation-dependent and may include such techniques as allocating space in a software-invisible buffer, allocating a cache line in only a single way, etc. For details, see the software-optimization documentation for a particular hardware implementation.
T0	All Cache Levels—Move the specified data into all cache levels.
T1	Level 2 and Higher—Move the specified data into all cache levels except 0th level (L1) cache.
T2	Level 3 and Higher—Move the specified data into all cache levels except 0th level (L1) and 1st level (L2) caches.

### Related Instructions

PREFETCH, PREFETCHW

**rFLAGS Affected**

None

**Exceptions**

None

**PUSH****Push onto Stack**

Decrements the stack pointer and then copies the specified immediate value or the value in the specified register or memory location to the top of the stack (the memory location pointed to by SS:rSP).

The operand-size attribute determines the number of bytes pushed to the stack. The stack-size attribute determines whether SP, ESP, or RSP is the stack pointer. The address-size attribute is used only to locate the memory operand when pushing a memory operand to the stack.

If the instruction pushes the stack pointer (rSP), the resulting value on the stack is that of rSP before execution of the instruction.

There is a PUSH CS instruction but no corresponding POP CS. The RET (Far) instruction pops a value from the top of stack into the CS register as part of its operation.

In 64-bit mode, the operand size of all PUSH instructions defaults to 64 bits, and there is no prefix available to encode a 32-bit operand size. Using the PUSH CS, PUSH DS, PUSH ES, or PUSH SS instructions in 64-bit mode generates an invalid-opcode exception.

Pushing an odd number of 16-bit operands when the stack address-size attribute is 32 results in a misaligned stack pointer.

Mnemonic	Opcode	Description
PUSH reg/mem16	FF /6	Push the contents of a 16-bit register or memory operand onto the stack.
PUSH reg/mem32	FF /6	Push the contents of a 32-bit register or memory operand onto the stack. (No prefix for encoding this in 64-bit mode.)
PUSH reg/mem64	FF /6	Push the contents of a 64-bit register or memory operand onto the stack.
PUSH reg16	50 +rw	Push the contents of a 16-bit register onto the stack.
PUSH reg32	50 +rd	Push the contents of a 32-bit register onto the stack. (No prefix for encoding this in 64-bit mode.)
PUSH reg64	50 +rq	Push the contents of a 64-bit register onto the stack.
PUSH imm8	6A ib	Push an 8-bit immediate value (sign-extended to 16, 32, or 64 bits) onto the stack.
PUSH imm16	68 iw	Push a 16-bit immediate value onto the stack.
PUSH imm32	68 id	Push a 32-bit immediate value onto the stack. (No prefix for encoding this in 64-bit mode.)
PUSH imm64	68 id	Push a sign-extended 32-bit immediate value onto the stack.
PUSH CS	0E	Push the CS selector onto the stack. (Invalid in 64-bit mode.)

Mnemonic	Opcode	Description
PUSH SS	16	Push the SS selector onto the stack. (Invalid in 64-bit mode.)
PUSH DS	1E	Push the DS selector onto the stack. (Invalid in 64-bit mode.)
PUSH ES	06	Push the ES selector onto the stack. (Invalid in 64-bit mode.)
PUSH FS	0F A0	Push the FS selector onto the stack.
PUSH GS	0F A8	Push the GS selector onto the stack.

## Related Instructions

POP

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	PUSH CS, PUSH DS, PUSH ES, or PUSH SS was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## PUSHA PUSHAD

### Push All GPRs onto Stack

Pushes the contents of the eAX, eCX, eDX, eBX, eSP (original value), eBP, eSI, and eDI general-purpose registers onto the stack in that order. This instruction decrements the stack pointer by 16 or 32 depending on operand size.

Using the PUSHA or PUSHAD instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
PUSHA	60	Push the contents of the AX, CX, DX, BX, original SP, BP, SI, and DI registers onto the stack. (Invalid in 64-bit mode.)
PUSHAD	60	Push the contents of the EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI registers onto the stack. (Invalid in 64-bit mode.)

### Related Instructions

POPA, POPAD

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## PUSHF PUSHFD PUSHFQ

### Push rFLAGS onto Stack

Decrements the rSP register and copies the rFLAGS register (except for the VM and RF flags) onto the stack. The instruction clears the VM and RF flags in the rFLAGS image before putting it on the stack.

The instruction pushes 2, 4, or 8 bytes, depending on the operand size.

In 64-bit mode, this instruction defaults to a 64-bit operand size and there is no prefix available to encode a 32-bit operand size.

In virtual-8086 mode, if system software has set the IOPL field to a value less than 3, a general-protection exception occurs if application software attempts to execute PUSHFx or POPFx while VME is not enabled or the operand size is not 16-bit.

Mnemonic	Opcode	Description
PUSHF	9C	Push the FLAGS word onto the stack.
PUSHFD	9C	Push the EFLAGS doubleword onto stack. (No prefix encoding this in 64-bit mode.)
PUSHFQ	9C	Push the RFLAGS quadword onto stack.

### Action

```
// See "Pseudocode Definition" on page 57.

PUSHF_START:
IF (REAL_MODE)
    PUSHF_REAL
ELSIF (PROTECTED_MODE)
    PUSHF_PROTECTED
ELSE // (VIRTUAL_MODE)
    PUSHF_VIRTUAL

PUSHF_REAL:
    PUSH.v old_RFLAGS      // Pushed with RF and VM cleared.
    EXIT

PUSHF_PROTECTED:
    PUSH.v old_RFLAGS      // Pushed with RF cleared.
    EXIT

PUSHF_VIRTUAL:
    IF (RFLAGS.IOPL==3)
    {
        PUSH.v old_RFLAGS // Pushed with RF,VM cleared.
        EXIT
    }
```

```

ELSIF ((CR4.VME==1) && (OPERAND_SIZE==16))
{
    PUSH.v old_RFLAGS // Pushed with VIF in the IF position.
                        // Pushed with IOPL=3.
    EXIT
}
ELSE // ((RFLAGS.IOPL<3) && ((CR4.VME==0) || (OPERAND_SIZE!=16)))
    EXCEPTION [#GP(0)]

```

## Related Instructions

POPF, POPFD, POPFQ

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP		X		The I/O privilege level was less than 3 and either VME was not enabled or the operand size was not 16-bit.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**RCL****Rotate Through Carry Left**

Rotates the bits of a register or memory location (first operand) to the left (more significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the right end (lsb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
<code>RCL reg/mem8, 1</code>	D0 /2	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left 1 bit.
<code>RCL reg/mem8, CL</code>	D2 /2	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified in the CL register.
<code>RCL reg/mem8, imm8</code>	C0 /2 ib	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.
<code>RCL reg/mem16, 1</code>	D1 /2	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left 1 bit.
<code>RCL reg/mem16, CL</code>	D3 /2	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified in the CL register.
<code>RCL reg/mem16, imm8</code>	C1 /2 ib	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.
<code>RCL reg/mem32, 1</code>	D1 /2	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left 1 bit.
<code>RCL reg/mem32, CL</code>	D3 /2	Rotate 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified in the CL register.
<code>RCL reg/mem32, imm8</code>	C1 /2 ib	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.
<code>RCL reg/mem64, 1</code>	D1 /2	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left 1 bit.

Mnemonic	Opcode	Description
RCL reg/mem64, CL	D3 /2	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified in the CL register.
RCL reg/mem64, imm8	C1 /2 ib	Rotates the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.

## Related Instructions

RCR, ROL, ROR

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**RCR****Rotate Through Carry Right**

Rotates the bits of a register or memory location (first operand) to the right (toward the less significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the left end (msb) of the first operand location.

The processor masks the upper three bits in the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the two most significant bits of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
<code>RCR reg/mem8, 1</code>	D0 /3	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right 1 bit.
<code>RCR reg/mem8,CL</code>	D2 /3	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified in the CL register.
<code>RCR reg/mem8,imm8</code>	C0 /3 ib	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value.
<code>RCR reg/mem16,1</code>	D1 /3	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right 1 bit.
<code>RCR reg/mem16,CL</code>	D3 /3	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified in the CL register.
<code>RCR reg/mem16, imm8</code>	C1 /3 ib	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value.
<code>RCR reg/mem32,1</code>	D1 /3	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right 1 bit.
<code>RCR reg/mem32,CL</code>	D3 /3	Rotate 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified in the CL register.
<code>RCR reg/mem32, imm8</code>	C1 /3 ib	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.
<code>RCR reg/mem64,1</code>	D1 /3	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right 1 bit.

Mnemonic	Opcode	Description
RCR <i>reg/mem64,CL</i>	D3 /3	Rotate 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified in the CL register.
RCR <i>reg/mem64, imm8</i>	C1 /3 ib	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified by an 8-bit immediate value.

## Related Instructions

RCL, ROR, ROL

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## RDFSBASE RDGSBASE

## Read FS.base Read GS.base

Copies the base field of the FS or GS segment descriptor to the specified register. When supported and enabled, these instructions can be executed at any processor privilege level. The RDFSBASE and RDGSBASE instructions are only defined in 64-bit mode.

System software must set the FSGSBASE bit (bit 16) of CR4 to enable the RDFSBASE and RDGSBASE instructions.

Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[FSGSBASE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
RDFSBASE <i>reg32</i>	F3 0F AE /0	Copy the lower 32 bits of FS.base to the specified general-purpose register.
RDFSBASE <i>reg64</i>	F3 0F AE /0	Copy the entire 64-bit contents of FS.base to the specified general-purpose register.
RDGSBASE <i>reg32</i>	F3 0F AE /1	Copy the lower 32 bits of GS.base to the specified general-purpose register.
RDGSBASE <i>reg64</i>	F3 0F AE /1	Copy the entire 64-bit contents of GS.base to the specified general-purpose register.

### Related Instructions

WRFSBASE, WRGSBASE

### rFLAGS Affected

None.

### Exceptions

Exception	Legacy	Compat- ibility	64-bit	Cause of Exception
#UD	X	X		Instruction is not valid in compatibility or legacy modes.
			X	Instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[FSGSBASE] = 0 or, if supported, not enabled in CR4.

## RDRAND

## Read Random

Loads the destination register with a hardware-generated random value.

The size of the returned value in bits is determined by the size of the destination register.

Hardware modifies the CF flag to indicate whether the value returned in the destination register is valid. If CF = 1, the value is valid. If CF = 0, the value is invalid. Software must test the state of the CF flag prior to using the value returned in the destination register to determine if the value is valid. If the returned value is invalid, software must execute the instruction again. Software should implement a retry limit to ensure forward progress of code.

The execution of RDRAND clears the OF, SF, ZF, AF, and PF flags.

Support for the RDRAND instruction is optional. On processors that support the instruction, CPUID Fn0000\_0001\_ECX[RDRAND] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
RDRAND reg16	0F C7 /6	Load the destination register with a 16-bit random number.
RDRAND reg32	0F C7 /6	Load the destination register with a 32-bit random number.
RDRAND reg64	0F C7 /6	Load the destination register with a 64-bit random number.

### Related Instructions

RDSEED

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn0000_0001_ECX[RDRAND] = 0.

## RDSEED

## Read Random Seed

Loads the destination register with a hardware-generated random “seed” value.

The size of the returned value in bits is determined by the size of the destination register.

Hardware modifies the CF flag to indicate whether the value returned in the destination register is valid. If CF = 1, the value is valid. If CF = 0, the value is invalid and will be returned as zero. Software must test the state of the CF flag prior to using the value returned in the destination register to determine if the value is valid. If the returned value is invalid, software must execute the instruction again. Software should implement a retry limit to ensure forward progress of code.

The execution of RDSEED clears the OF, SF, ZF, AF, and PF flags.

Mnemonic	Opcode	Description
RDSEED reg16	0F C7 /7	Read 16-bit random seed
RDSEED reg32	0F C7 /7	Read 32-bit random seed
RDSEED reg64	0F C7 /7	Read 64-bit random seed

### Related Instructions

RDRAND

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[RDSEED] = 0

## RET (Near)

## Near Return from Called Procedure

Returns from a procedure previously entered by a CALL near instruction. This form of the RET instruction returns to a calling procedure within the current code segment.

This instruction pops the rIP from the stack, with the size of the pop determined by the operand size. The new rIP is then zero-extended to 64 bits. The RET instruction can accept an immediate value operand that it adds to the rSP after it pops the target rIP. This action skips over any parameters previously passed back to the subroutine that are no longer needed.

In 64-bit mode, the operand size defaults to 64 bits (eight bytes) without the need for a REX prefix. No prefix is available to encode a 32-bit operand size in 64-bit mode.

See RET (Far) for information on far returns—returns to procedures located outside of the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
RET	C3	Near return to the calling procedure.
RET <i>imm16</i>	C2 <i>iw</i>	Near return to the calling procedure then pop the specified number of bytes from the stack.

### Related Instructions

CALL (Near), CALL (Far), RET (Far)

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## RET (Far)

## Far Return from Called Procedure

Returns from a procedure previously entered by a CALL Far instruction. This form of the RET instruction returns to a calling procedure in a different segment than the current code segment. It can return to the same CPL or to a less privileged CPL.

RET Far pops a target CS and rIP from the stack. If the new code segment is less privileged than the current code segment, the stack pointer is incremented by the number of bytes indicated by the immediate operand, if present; then a new SS and rSP are also popped from the stack.

The final value of rSP is incremented by the number of bytes indicated by the immediate operand, if present. This action skips over the parameters (previously passed to the subroutine) that are no longer needed.

All stack pops are determined by the operand size. If necessary, the target rIP is zero-extended to 64 bits before assuming program control.

If the CPL changes, the data segment selectors are set to NULL for any of the data segments (DS, ES, FS, GS) not accessible at the new CPL.

See RET (Near) for information on near returns—returns to procedures located inside the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
RETF	CB	Far return to the calling procedure.
RETF <i>imm16</i>	CA <i>iw</i>	Far return to the calling procedure, then pop the specified number of bytes from the stack.

### Action

```

// Far returns (RETF)
// See "Pseudocode Definition" on page 57.

RETF_START:

IF (REAL_MODE)
    RETF_REAL_OR_VIRTUAL
ELSIF (PROTECTED_MODE)
    RETF_PROTECTED
ELSE // (VIRTUAL_MODE)
    RETF_REAL_OR_VIRTUAL

RETF_REAL_OR_VIRTUAL:

IF (OPCODE == retf imm16)
    temp_IMM = word-sized immediate specified in the instruction,
                zero-extended to 64 bits

```

```

ELSE // (OPCODE == retf)
    temp_IMM = 0

    POP.v temp_RIP
    POP.v temp_CS

    IF (temp_RIP > CS.limit)
        EXCEPTION [#GP(0)]

    CS.sel = temp_CS
    CS.base = temp_CS SHL 4

    RSP.s = RSP + temp_IMM
    RIP = temp_RIP
    EXIT

RETF_PROTECTED:

IF (OPCODE == retf imm16)
    temp_IMM = word-sized immediate specified in the instruction,
               zero-extended to 64 bits
ELSE // (OPCODE == retf)
    temp_IMM = 0

    POP.v temp_RIP
    POP.v temp_CS

    temp_CPL = temp_CS.rpl

    IF (CPL==temp_CPL)
    {
        CS = READ_DESCRIPTOR (temp_CS, iret_chk)

        RSP.s = RSP + temp_IMM

        IF ((64BIT_MODE) && (temp_RIP is non-canonical)
            || (!64BIT_MODE) && (temp_RIP > CS.limit))
            EXCEPTION [#GP(0)]

        RIP = temp_RIP
        EXIT
    }
ELSE // (CPL!=temp_CPL)
{
    RSP.s = RSP + temp_IMM

    POP.v temp_RSP
    POP.v temp_SS

    CS = READ_DESCRIPTOR (temp_CS, iret_chk)
}

```

```

CPL = temp_CPL

IF (((64BIT_MODE) && (temp_RIP is non-canonical)
    || (!64BIT_MODE) && (temp_RIP > CS.limit))
    EXCEPTION [#GP(0)]

SS = READ_DESCRIPTOR (temp_SS, ss_chk)

RSP.s = temp_RSP + temp_IMM

IF (changing CPL)
{
    FOR (seg = ES, DS, FS, GS)
        IF ((seg.attr.dpl < CPL) && ((seg.attr.type == 'data'
            || (seg.attr.type == 'non-conforming-code'))))
        {
            seg = NULL // can't use lower dpl data segment at higher cpl
        }
}

RIP = temp_RIP
EXIT
}

```

## Related Instructions

CALL (Near), CALL (Far), RET (Near)

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Segment not present, #NP (selector)			X	The return code segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The return stack segment was marked not present.
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)		X		The return code selector was a null selector.
		X		The return stack selector was a null selector and the return mode was non-64-bit mode or CPL was 3.
		X		The return code or stack descriptor exceeded the descriptor table limit.
		X		The return code or stack selector's TI bit was set but the LDT selector was a null selector.
		X		The segment descriptor for the return code was not a code segment.
		X		The RPL of the return code segment selector was less than the CPL.
		X		The return code segment was non-conforming and the segment selector's DPL was not equal to the RPL of the code segment's segment selector.
		X		The return code segment was conforming and the segment selector's DPL was greater than the RPL of the code segment's segment selector.
		X		The segment descriptor for the return stack was not a writable data segment.
		X		The stack segment descriptor DPL was not equal to the RPL of the return code segment selector.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned-memory reference was performed while alignment checking was enabled.

## ROL

## Rotate Left

Rotates the bits of a register or memory location (first operand) to the left (toward the more significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out left are rotated back in at the right end (lsb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, it masks the upper two bits of the count, providing a count in the range of 0 to 63.

After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the lsb of the result). For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
ROL reg/mem8, 1	D0 /0	Rotate an 8-bit register or memory operand left 1 bit.
ROL reg/mem8, CL	D2 /0	Rotate an 8-bit register or memory operand left the number of bits specified in the CL register.
ROL reg/mem8, imm8	C0 /0 ib	Rotate an 8-bit register or memory operand left the number of bits specified by an 8-bit immediate value.
ROL reg/mem16, 1	D1 /0	Rotate a 16-bit register or memory operand left 1 bit.
ROL reg/mem16, CL	D3 /0	Rotate a 16-bit register or memory operand left the number of bits specified in the CL register.
ROL reg/mem16, imm8	C1 /0 ib	Rotate a 16-bit register or memory operand left the number of bits specified by an 8-bit immediate value.
ROL reg/mem32, 1	D1 /0	Rotate a 32-bit register or memory operand left 1 bit.
ROL reg/mem32, CL	D3 /0	Rotate a 32-bit register or memory operand left the number of bits specified in the CL register.
ROL reg/mem32, imm8	C1 /0 ib	Rotate a 32-bit register or memory operand left the number of bits specified by an 8-bit immediate value.
ROL reg/mem64, 1	D1 /0	Rotate a 64-bit register or memory operand left 1 bit.
ROL reg/mem64, CL	D3 /0	Rotate a 64-bit register or memory operand left the number of bits specified in the CL register.
ROL reg/mem64, imm8	C1 /0 ib	Rotate a 64-bit register or memory operand left the number of bits specified by an 8-bit immediate value.

## Related Instructions

RCL, RCR, ROR

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**ROR****Rotate Right**

Rotates the bits of a register or memory location (first operand) to the right (toward the less significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out right are rotated back in at the left end (the most significant bit) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the most significant bit of the result). For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the two most significant bits of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
<code>ROR reg/mem8, 1</code>	D0 /1	Rotate an 8-bit register or memory location right 1 bit.
<code>ROR reg/mem8, CL</code>	D2 /1	Rotate an 8-bit register or memory location right the number of bits specified in the CL register.
<code>ROR reg/mem8, imm8</code>	C0 /1 ib	Rotate an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value.
<code>ROR reg/mem16, 1</code>	D1 /1	Rotate a 16-bit register or memory location right 1 bit.
<code>ROR reg/mem16, CL</code>	D3 /1	Rotate a 16-bit register or memory location right the number of bits specified in the CL register.
<code>ROR reg/mem16, imm8</code>	C1 /1 ib	Rotate a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value.
<code>ROR reg/mem32, 1</code>	D1 /1	Rotate a 32-bit register or memory location right 1 bit.
<code>ROR reg/mem32, CL</code>	D3 /1	Rotate a 32-bit register or memory location right the number of bits specified in the CL register.
<code>ROR reg/mem32, imm8</code>	C1 /1 ib	Rotate a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.
<code>ROR reg/mem64, 1</code>	D1 /1	Rotate a 64-bit register or memory location right 1 bit.
<code>ROR reg/mem64, CL</code>	D3 /1	Rotate a 64-bit register or memory operand right the number of bits specified in the CL register.
<code>ROR reg/mem64, imm8</code>	C1 /1 ib	Rotate a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.

**Related Instructions**

`RCL`, `RCR`, `ROL`

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## RORX

## Rotate Right Extended

Rotates the bits of the source operand right (toward the least-significant bit) by the number of bit positions specified in an immediate operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

**RORX dest, src, rot\_cnt**

On each right-shift, the bit shifted out of the least-significant bit position is copied to the most-significant bit. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the source (*src*) is either a general-purpose register or a memory operand. The rotate count *rot\_cnt* is encoded in an immediate byte. When the operand size is 32, bits [7:5] of the immediate byte are ignored; when the operand size is 64, bits [7:6] of the immediate byte are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
RORX reg32, reg/mem32, imm8	C4	$\overline{\text{RXB}}.03$	0.1111.0.11	F0 /r ib
RORX reg64, reg/mem64, imm8	C4	$\overline{\text{RXB}}.03$	1.1111.0.11	F0 /r ib

### Related Instructions

SARX, SHLX, SHRX

### rFLAGS Affected

None.

## Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## SAHF

## Store AH into Flags

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). The instruction ignores bits 1, 3, and 5 of register AH; it sets those bits in the EFLAGS register to 1, 0, and 0, respectively.

The SAHF instruction can only be executed in 64-bit mode. Support for the instruction is implementation-dependent and is indicated by CPUID Fn8000\_0001\_ECX[LahfSahf] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
SAHF	9E	Loads the sign flag, the zero flag, the auxiliary flag, the parity flag, and the carry flag from the AH register into the lower 8 bits of the EFLAGS register.

## Related Instructions

LAHF

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
												M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	The SAHF instruction is not supported, as indicated by CPUID Fn8000_0001_ECX[LahfSahf] = 0.

**SAL  
SHL****Shift Left**

Shifts the bits of a register or memory location (first operand) to the left through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. For each bit shift, the SAL instruction clears the least-significant bit to 0. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

The effect of this instruction is multiplication by powers of two.

For 1-bit shifts, the instruction sets the OF flag to the logical `xor` of the CF bit (after the shift) and the most significant bit of the result. When the shift count is greater than 1, the OF flag is undefined.

If the shift count is 0, no flags are modified.

SHL is an alias to the SAL instruction.

Mnemonic	Opcode	Description
SAL reg/mem8, 1	D0 /4	Shift an 8-bit register or memory location left 1 bit.
SAL reg/mem8, CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.
SAL reg/mem8, imm8	C0 /4 ib	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SAL reg/mem16, 1	D1 /4	Shift a 16-bit register or memory location left 1 bit.
SAL reg/mem16, CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.
SAL reg/mem16, imm8	C1 /4 ib	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SAL reg/mem32, 1	D1 /4	Shift a 32-bit register or memory location left 1 bit.
SAL reg/mem32, CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.
SAL reg/mem32, imm8	C1 /4 ib	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SAL reg/mem64, 1	D1 /4	Shift a 64-bit register or memory location left 1 bit.
SAL reg/mem64, CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.
SAL reg/mem64, imm8	C1 /4 ib	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.

Mnemonic	Opcode	Description
SHL reg/mem8, 1	D0 /4	Shift an 8-bit register or memory location by 1 bit.
SHL reg/mem8, CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.
SHL reg/mem8, imm8	C0 /4 ib	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SHL reg/mem16, 1	D1 /4	Shift a 16-bit register or memory location left 1 bit.
SHL reg/mem16, CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.
SHL reg/mem16, imm8	C1 /4 ib	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SHL reg/mem32, 1	D1 /4	Shift a 32-bit register or memory location left 1 bit.
SHL reg/mem32, CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.
SHL reg/mem32, imm8	C1 /4 ib	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SHL reg/mem64, 1	D1 /4	Shift a 64-bit register or memory location left 1 bit.
SHL reg/mem64, CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.
SHL reg/mem64, imm8	C1 /4 ib	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.

## Related Instructions

SAR, SHR, SHLD, SHRD

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS		X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## SAR

## Shift Arithmetic Right

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The SAR instruction does not change the sign bit of the target operand. For each bit shift, it copies the sign bit to the next bit, preserving the sign of the result.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit shifts, the instruction clears the OF flag to 0. When the shift count is greater than 1, the OF flag is undefined.

If the shift count is 0, no flags are modified.

Although the SAR instruction effectively divides the operand by a power of 2, the behavior is different from the IDIV instruction. For example, shifting –11 (FFFFFFF5h) by two bits to the right (that is, divide –11 by 4), gives a result of FFFFFFFDh, or –3, whereas the IDIV instruction for dividing –11 by 4 gives a result of –2. This is because the IDIV instruction rounds off the quotient to zero, whereas the SAR instruction rounds off the remainder to zero for positive dividends and to negative infinity for negative dividends. So, for positive operands, SAR behaves like the corresponding IDIV instruction. For negative operands, it gives the same result if and only if all the shifted-out bits are zeroes; otherwise, the result is smaller by 1.

Mnemonic	Opcode	Description
SAR reg/mem8, 1	D0 /7	Shift a signed 8-bit register or memory operand right 1 bit.
SAR reg/mem8, CL	D2 /7	Shift a signed 8-bit register or memory operand right the number of bits specified in the CL register.
SAR reg/mem8, imm8	C0 /7 ib	Shift a signed 8-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SAR reg/mem16, 1	D1 /7	Shift a signed 16-bit register or memory operand right 1 bit.
SAR reg/mem16, CL	D3 /7	Shift a signed 16-bit register or memory operand right the number of bits specified in the CL register.
SAR reg/mem16, imm8	C1 /7 ib	Shift a signed 16-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SAR reg/mem32, 1	D1 /7	Shift a signed 32-bit register or memory location 1 bit.
SAR reg/mem32, CL	D3 /7	Shift a signed 32-bit register or memory location right the number of bits specified in the CL register.

Mnemonic	Opcode	Description
SAR reg/mem32, imm8	C1 /7 ib	Shift a signed 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.
SAR reg/mem64, 1	D1 /7	Shift a signed 64-bit register or memory location right 1 bit.
SAR reg/mem64, CL	D3 /7	Shift a signed 64-bit register or memory location right the number of bits specified in the CL register.
SAR reg/mem64, imm8	C1 /7 ib	Shift a signed 64-bit register or memory location right the number of bits specified by an 8-bit immediate value.

## Related Instructions

SAL, SHL, SHR, SHLD, SHRD

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## SARX

## Shift Right Arithmetic Extended

Shifts the bits of the first source operand right (toward the least-significant bit) arithmetically by the number of bit positions specified in the second source operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

**SARX dest, src, shft\_cnt**

On each right-shift, the most-significant bit (the sign bit) is replicated. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the first source (*src*) is either a general-purpose register or a memory operand. The second source operand *shft\_cnt* is a general-purpose register. When the operand size is 32, bits [31:5] of *shft\_cnt* are ignored; when the operand size is 64, bits [63:6] of *shft\_cnt* are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SARX reg32, reg/mem32, reg32	C4	$\overline{\text{RXB}}.02$	0. $\overline{\text{src}}$ 2.0.10	F7 /r
SARX reg64, reg/mem64, reg64	C4	$\overline{\text{RXB}}.02$	1. $\overline{\text{src}}$ 2.0.10	F7 /r

### Related Instructions

RORX, SHLX, SHRX

### rFLAGS Affected

None.

## Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**SBB****Subtract with Borrow**

Subtracts an immediate value or the value in a register or a memory location (second operand) from a register or a memory location (first operand), and stores the result in the first operand location. If the carry flag (CF) is 1, the instruction subtracts 1 from the result. Otherwise, it operates like SUB.

The SBB instruction sign-extends immediate value operands to the length of the first operand size.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

This instruction is useful for multibyte (multiword) numbers because it takes into account the borrow from a previous SUB instruction.

The forms of the SBB instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
SBB AL, <i>imm8</i>	1C <i>ib</i>	Subtract an immediate 8-bit value from the AL register with borrow.
SBB AX, <i>imm16</i>	1D <i>iw</i>	Subtract an immediate 16-bit value from the AX register with borrow.
SBB EAX, <i>imm32</i>	1D <i>id</i>	Subtract an immediate 32-bit value from the EAX register with borrow.
SBB RAX, <i>imm32</i>	1D <i>id</i>	Subtract a sign-extended immediate 32-bit value from the RAX register with borrow.
SBB <i>reg/mem8, imm8</i>	80 /3 <i>ib</i>	Subtract an immediate 8-bit value from an 8-bit register or memory location with borrow.
SBB <i>reg/mem16, imm16</i>	81 /3 <i>iw</i>	Subtract an immediate 16-bit value from a 16-bit register or memory location with borrow.
SBB <i>reg/mem32, imm32</i>	81 /3 <i>id</i>	Subtract an immediate 32-bit value from a 32-bit register or memory location with borrow.
SBB <i>reg/mem64, imm32</i>	81 /3 <i>id</i>	Subtract a sign-extended immediate 32-bit value from a 64-bit register or memory location with borrow.
SBB <i>reg/mem16, imm8</i>	83 /3 <i>ib</i>	Subtract a sign-extended 8-bit immediate value from a 16-bit register or memory location with borrow.
SBB <i>reg/mem32, imm8</i>	83 /3 <i>ib</i>	Subtract a sign-extended 8-bit immediate value from a 32-bit register or memory location with borrow.
SBB <i>reg/mem64, imm8</i>	83 /3 <i>ib</i>	Subtract a sign-extended 8-bit immediate value from a 64-bit register or memory location with borrow.
SBB <i>reg/mem8, reg8</i>	18 / <i>r</i>	Subtract the contents of an 8-bit register from an 8-bit register or memory location with borrow.
SBB <i>reg/mem16, reg16</i>	19 / <i>r</i>	Subtract the contents of a 16-bit register from a 16-bit register or memory location with borrow.

Mnemonic	Opcode	Description
SBB reg/mem32, reg32	19 /r	Subtract the contents of a 32-bit register from a 32-bit register or memory location with borrow.
SBB reg/mem64, reg64	19 /r	Subtract the contents of a 64-bit register from a 64-bit register or memory location with borrow.
SBB reg8, reg/mem8	1A /r	Subtract the contents of an 8-bit register or memory location from the contents of an 8-bit register with borrow.
SBB reg16, reg/mem16	1B /r	Subtract the contents of a 16-bit register or memory location from the contents of a 16-bit register with borrow.
SBB reg32, reg/mem32	1B /r	Subtract the contents of a 32-bit register or memory location from the contents of a 32-bit register with borrow.
SBB reg64, reg/mem64	1B /r	Subtract the contents of a 64-bit register or memory location from the contents of a 64-bit register with borrow.

## Related Instructions

SUB, ADD, ADC

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## **SCAS** **SCASB** **SCASW** **SCASD** **SCASQ**

## **Scan String**

Compares the AL, AX, EAX, or RAX register with the byte, word, doubleword, or quadword pointed to by ES:rDI, sets the status flags in the rFLAGS register according to the results, and then increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments the rDI register; otherwise, it decrements it. The instruction increments or decrements the rDI register by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the SCASx instruction with an explicit operand address the operand at ES:rDI. The explicit operand serves only to specify the size of the values being compared.

The no-operands forms of the instruction use the ES:rDI registers to point to the value to be compared. The mnemonic determines the size of the operands and the specific register containing the other comparison value.

For block comparisons, the SCASx instructions support the REPE or REPZ prefixes (they are synonyms) and the REPNE or REPNZ prefixes (they are synonyms). For details about the REP prefixes, see “Repeat Prefixes” on page 12. A SCASx instruction can also operate inside a loop controlled by the LOOPcc instruction.

Mnemonic	Opcode	Description
SCAS <i>mem8</i>	AE	Compare the contents of the AL register with the byte at ES:rDI, and then increment or decrement rDI.
SCAS <i>mem16</i>	AF	Compare the contents of the AX register with the word at ES:rDI, and then increment or decrement rDI.
SCAS <i>mem32</i>	AF	Compare the contents of the EAX register with the doubleword at ES:rDI, and then increment or decrement rDI.
SCAS <i>mem64</i>	AF	Compare the contents of the RAX register with the quadword at ES:rDI, and then increment or decrement rDI.
SCASB	AE	Compare the contents of the AL register with the byte at ES:rDI, and then increment or decrement rDI.
SCASW	AF	Compare the contents of the AX register with the word at ES:rDI, and then increment or decrement rDI.

Mnemonic	Opcode	Description
SCASD	AF	Compare the contents of the EAX register with the doubleword at ES:rDI, and then increment or decrement rDI.
SCASQ	AF	Compare the contents of the RAX register with the quadword at ES:rDI, and then increment or decrement rDI.

## Related Instructions

CMP, CMPS<sub>x</sub>

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP			X	A null ES segment was used to reference memory.
	X	X	X	A memory address exceeded the ES segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## SETcc

## Set Byte on Condition

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified in the mnemonic (*cc*), sets the value in the specified 8-bit memory location or register to 1. If the flags do not meet the specified condition, SETcc clears the memory location or register to 0.

Mnemonics with the A (above) and B (below) tags are intended for use when performing unsigned integer comparisons; those with G (greater) and L (less) tags are intended for use with signed integer comparisons.

Software typically uses the SETcc instructions to set logical indicators. Like the CMOVcc instructions (page 145), the SETcc instructions can replace two instructions—a conditional jump and a move. Replacing conditional jumps with conditional sets can help avoid branch-prediction penalties that may result from conditional jumps.

If the logical value “true” (logical one) is represented in a high-level language as an integer with all bits set to 1, software can accomplish such representation by first executing the opposite SETcc instruction—for example, the opposite of SETZ is SETNZ—and then decrementing the result.

A ModR/M byte is used to identify the operand. The *reg* field in the ModR/M byte is unused.

Mnemonic	Opcode	Description
SETO <i>reg/mem8</i>	0F 90 /0	Set byte if overflow (OF = 1).
SETNO <i>reg/mem8</i>	0F 91 /0	Set byte if not overflow (OF = 0).
SETB <i>reg/mem8</i>	0F 92 /0	Set byte if below (CF = 1).
SETC <i>reg/mem8</i>		Set byte if carry (CF = 1).
SETNAE <i>reg/mem8</i>		Set byte if not above or equal (CF = 1).
SETNB <i>reg/mem8</i>	0F 93 /0	Set byte if not below (CF = 0).
SETNC <i>reg/mem8</i>		Set byte if not carry (CF = 0).
SETAE <i>reg/mem8</i>		Set byte if above or equal (CF = 0).
SETZ <i>reg/mem8</i>	0F 94 /0	Set byte if zero (ZF = 1).
SETE <i>reg/mem8</i>		Set byte if equal (ZF = 1).
SETNZ <i>reg/mem8</i>	0F 95 /0	Set byte if not zero (ZF = 0).
SETNE <i>reg/mem8</i>		Set byte if not equal (ZF = 0).
SETBE <i>reg/mem8</i>	0F 96 /0	Set byte if below or equal (CF = 1 or ZF = 1).
SETNA <i>reg/mem8</i>		Set byte if not above (CF = 1 or ZF = 1).
SETNBE <i>reg/mem8</i>	0F 97 /0	Set byte if not below or equal (CF = 0 and ZF = 0).
SETA <i>reg/mem8</i>		Set byte if above (CF = 0 and ZF = 0).
SETS <i>reg/mem8</i>	0F 98 /0	Set byte if sign (SF = 1).
SETNS <i>reg/mem8</i>	0F 99 /0	Set byte if not sign (SF = 0).
SETP <i>reg/mem8</i>	0F 9A /0	Set byte if parity (PF = 1).
SETPE <i>reg/mem8</i>		Set byte if parity even (PF = 1).
SETNP <i>reg/mem8</i>	0F 9B /0	Set byte if not parity (PF = 0).
SETPO <i>reg/mem8</i>		Set byte if parity odd (PF = 0).

Mnemonic	Opcode	Description
SETL <i>reg/mem8</i>	0F 9C /0	Set byte if less (SF <> OF).
SETNGE <i>reg/mem8</i>		Set byte if not greater or equal (SF <> OF).
SETNL <i>reg/mem8</i>	0F 9D /0	Set byte if not less (SF = OF).
SETGE <i>reg/mem8</i>		Set byte if greater or equal (SF = OF).
SETLE <i>reg/mem8</i>	0F 9E /0	Set byte if less or equal (ZF = 1 or SF <> OF).
SETNG <i>reg/mem8</i>		Set byte if not greater (ZF = 1 or SF <> OF).
SETNLE <i>reg/mem8</i>	0F 9F /0	Set byte if not less or equal (ZF = 0 and SF = OF).
SETG <i>reg/mem8</i>		Set byte if greater (ZF = 0 and SF = OF).

## Related Instructions

None

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

## SFENCE

## Store Fence

Acts as a barrier to force strong memory ordering (serialization) between store instructions preceding the SFENCE and store instructions that follow the SFENCE. Stores to differing memory types, or within the WC memory type, may become visible out of program order; the SFENCE instruction ensures that the system completes all previous stores in such a way that they are globally visible before executing subsequent stores. This includes emptying the store buffer and all write-combining buffers.

The SFENCE instruction is weakly-ordered with respect to load instructions, data and instruction prefetches, and the LFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an SFENCE.

In addition to store instructions, SFENCE is strongly ordered with respect to other SFENCE instructions, MFENCE instructions, and serializing instructions. Further details on the use of MFENCE to order accesses among differing memory types may be found in *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*, section 7.4 “Memory Types” on page 172.

The SFENCE instruction is an SSE1 instruction. Support for SSE1 instructions is indicated by CPUID Fn0000\_0001\_EDX[SSE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
SFENCE	0F AE F8	Force strong ordering of (serialized) store operations.

### Related Instructions

LFENCE, MFENCE

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid Opcode, #UD	X	X	X	The SSE instructions are not supported, as indicated by EDX bit 25 of CPUID function 0000_0001h; and the AMD extensions to MMX are not supported, as indicated by EDX bit 22 of CPUID function 8000_0001h.

**SHL****Shift Left**

This instruction is synonymous with the SAL instruction. For information, see “SAL SHL” on page 301.

## SHLD

## Shift Left Double

Shifts the bits of a register or memory location (first operand) to the left by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the right. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. If the masked count is greater than the operand size, the result in the destination register is undefined.

If the shift count is 0, no flags are modified.

If the count is 1 and the sign of the operand being shifted changes, the instruction sets the OF flag to 1. If the count is greater than 1, OF is undefined.

Mnemonic	Opcode	Description
SHLD <i>reg/mem16, reg16, imm8</i>	0F A4 /r ib	Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHLD <i>reg/mem16, reg16, CL</i>	0F A5 /r	Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.
SHLD <i>reg/mem32, reg32, imm8</i>	0F A4 /r ib	Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHLD <i>reg/mem32, reg32, CL</i>	0F A5 /r	Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.
SHLD <i>reg/mem64, reg64, imm8</i>	0F A4 /r ib	Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHLD <i>reg/mem64, reg64, CL</i>	0F A5 /r	Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.

## Related Instructions

SHRD, SAL, SAR, SHR, SHL

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## SHLX

## Shift Left Logical Extended

Shifts the bits of the first source operand left (toward the most-significant bit) by the number of bit positions specified in the second source operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

**SHLX dest, src, shft\_cnt**

On each left-shift, a zero is shifted into the least-significant bit position. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the first source (*src*) is either a general-purpose register or a memory operand. The second source operand *shft\_cnt* is a general-purpose register. When the operand size is 32, bits [31:5] of *shft\_cnt* are ignored; when the operand size is 64, bits [63:6] of *shft\_cnt* are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SHLX reg32, reg/mem32, reg32	C4	$\overline{\text{RXB}}.02$	0. $\overline{\text{src}}$ 2.0.01	F7 /r
SHLX reg64, reg/mem64, reg64	C4	$\overline{\text{RXB}}.02$	1. $\overline{\text{src}}$ 2.0.01	F7 /r

### Related Instructions

RORX, SARX, SHRX

### rFLAGS Affected

None.

## Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**SHR****Shift Right**

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

For each bit shift, the instruction clears the most-significant bit to 0.

The effect of this instruction is unsigned division by powers of two.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit shifts, the instruction sets the OF flag to the most-significant bit of the original value. If the count is greater than 1, the OF flag is undefined.

If the shift count is 0, no flags are modified.

Mnemonic	Opcode	Description
SHR reg/mem8, 1	D0 /5	Shift an 8-bit register or memory operand right 1 bit.
SHR reg/mem8, CL	D2 /5	Shift an 8-bit register or memory operand right the number of bits specified in the CL register.
SHR reg/mem8, imm8	C0 /5 ib	Shift an 8-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SHR reg/mem16, 1	D1 /5	Shift a 16-bit register or memory operand right 1 bit.
SHR reg/mem16, CL	D3 /5	Shift a 16-bit register or memory operand right the number of bits specified in the CL register.
SHR reg/mem16, imm8	C1 /5 ib	Shift a 16-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SHR reg/mem32, 1	D1 /5	Shift a 32-bit register or memory operand right 1 bit.
SHR reg/mem32, CL	D3 /5	Shift a 32-bit register or memory operand right the number of bits specified in the CL register.
SHR reg/mem32, imm8	C1 /5 ib	Shift a 32-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SHR reg/mem64, 1	D1 /5	Shift a 64-bit register or memory operand right 1 bit.
SHR reg/mem64, CL	D3 /5	Shift a 64-bit register or memory operand right the number of bits specified in the CL register.
SHR reg/mem64, imm8	C1 /5 ib	Shift a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.

## Related Instructions

SHL, SAL, SAR, SHLD, SHRD

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## SHRD

## Shift Right Double

Shifts the bits of a register or memory location (first operand) to the right by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the left. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. If the masked count is greater than the operand size, the result in the destination register is undefined.

If the shift count is 0, no flags are modified.

If the count is 1 and the sign of the value being shifted changes, the instruction sets the OF flag to 1. If the count is greater than 1, the OF flag is undefined.

Mnemonic	Opcode	Description
SHRD reg/mem16, reg16, imm8	0F AC /r ib	Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHRD reg/mem16, reg16, CL	0F AD /r	Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.
SHRD reg/mem32, reg32, imm8	0F AC /r ib	Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHRD reg/mem32, reg32, CL	0F AD /r	Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.
SHRD reg/mem64, reg64, imm8	0F AC /r ib	Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHRD reg/mem64, reg64, CL	0F AD /r	Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.

## Related Instructions

SHLD, SHR, SHL, SAR, SAL

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## SHRX

## Shift Right Logical Extended

Shifts the bits of the first source operand right (toward the least-significant bit) by the number of bit positions specified in the second source operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

**SHRX *dest, src, shft\_cnt***

On each right-shift, a zero is shifted into the most-significant bit position. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the first source (*src*) is either a general-purpose register or a memory operand. The second source operand *shft\_cnt* is a general-purpose register. When the operand size is 32, bits [31:5] of *shft\_cnt* are ignored; when the operand size is 64, bits [63:6] of *shft\_cnt* are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SHRX <i>reg32, reg/mem32, reg32</i>	C4	$\overline{\text{RXB}}.02$	0. $\overline{\text{src2}}$ .0.11	F7 /r
SHRX <i>reg64, reg/mem64, reg64</i>	C4	$\overline{\text{RXB}}.02$	1. $\overline{\text{src2}}$ .0.11	F7 /r

### Related Instructions

RORX, SARX, SHLX

### rFLAGS Affected

None.

## Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**SLWPCB****Store Lightweight Profiling Control Block Address**

Flushes Lightweight Profiling (LWP) state to memory and returns the current effective address of the Lightweight Profiling Control Block (LWPCB) in the specified register. The LWPCB address returned is truncated to 32 bits if the operand size is 32.

If LWP is not currently enabled, SLWPCB sets the specified register to zero.

The flush operation stores the internal event counters for active events and the current ring buffer head pointer into the LWPCB. If there is an unwritten event record pending, it is written to the event ring buffer.

The LWP\_CBAADDR MSR holds the linear address of the current LWPCB. If the contents of LWP\_CBAADDR is not zero, the value returned in the specified register is an effective address that is calculated by subtracting the current DS.Base address from the linear address kept in LWP\_CBAADDR. Note that if DS has changed between the time LLWPCB was executed and the time SLWPCB is executed, this might result in an address that is not currently accessible by the application.

SLWPCB generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

It is possible to execute SLWPCB when the CPL != 3 or when SMM is active, but if the LWPCB pointer is not zero, system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF fault. Using SLWPCB in these situations is not recommended.

See the discussion of lightweight profiling in Volume 2, Chapter 13 for more information on the use of the LLWPCB, SLWPCB, LWPINS, and LWPVAL instructions.

The SLWPCB instruction is implemented if LWP is supported on a processor. Support for LWP is indicated by CPUID Fn8000\_0001\_ECX[LWP] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

**Instruction Encoding**

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
SLWPCB reg32	8F	<u>RXB</u> .09	0.1111.0.00	12 /1
SLWPCB reg64	8F	<u>RXB</u> .09	1.1111.0.00	12 /1

ModRM.reg augments the opcode and is assigned the value 001b. ModRM.r/m (augmented by XOP.R) specifies the register in which to put the LWPCB address. ModRM.mod must be 11b.

## Related Instructions

LLWPCB, LWPINS, LWPVAL

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SLWPCB instruction is not supported, as indicated by CPUID Fn8000_0001_ECX[LWP] = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available, or mod != 11b, or vvvv != 1111b.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from flushing an event to the ring buffer.

## STC

## Set Carry Flag

Sets the carry flag (CF) in the rFLAGS register to one.

Mnemonic	Opcode	Description
STC	F9	Set the carry flag (CF) to one.

### Related Instructions

CLC, CMC

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																1
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

### Exceptions

None

**STD****Set Direction Flag**

Set the direction flag (DF) in the rFLAGS register to 1. If the DF flag is 0, each iteration of a string instruction increments the data pointer (index registers rSI or rDI). If the DF flag is 1, the string instruction decrements the pointer. Use the CLD instruction before a string instruction to make the data pointer increment.

Mnemonic	Opcode	Description
STD	FD	Set the direction flag (DF) to one.

**Related Instructions**

CLD, INSx, LODSx, MOVSx, OUTSx, SCASx, STOSx, CMPSx

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
									1							
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

None

**STOS****STOSB****STOSW****STOSD****STOSQ****Store String**

Copies a byte, word, doubleword, or quadword from the AL, AX, EAX, or RAX registers to the memory location pointed to by ES:rDI and increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments the pointer; otherwise, it decrements the pointer. It increments or decrements the pointer by 1, 2, 4, or 8, depending on the size of the value being copied.

The forms of the STOSx instruction with an explicit operand use the operand only to specify the type (size) of the value being copied.

The no-operands forms specify the type (size) of the value being copied with the mnemonic.

The STOSx instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 12. The STOSx instructions can also operate inside a LOOPcc instruction.

Mnemonic	Opcode	Description
STOS <i>mem8</i>	AA	Store the contents of the AL register to ES:rDI, and then increment or decrement rDI.
STOS <i>mem16</i>	AB	Store the contents of the AX register to ES:rDI, and then increment or decrement rDI.
STOS <i>mem32</i>	AB	Store the contents of the EAX register to ES:rDI, and then increment or decrement rDI.
STOS <i>mem64</i>	AB	Store the contents of the RAX register to ES:rDI, and then increment or decrement rDI.
STOSB	AA	Store the contents of the AL register to ES:rDI, and then increment or decrement rDI.
STOSW	AB	Store the contents of the AX register to ES:rDI, and then increment or decrement rDI.
STOSD	AB	Store the contents of the EAX register to ES:rDI, and then increment or decrement rDI.
STOSQ	AB	Store the contents of the RAX register to ES:rDI, and then increment or decrement rDI.

**Related Instructions**

LODSx, MOVSx

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	A memory address exceeded the ES segment limit or was non-canonical.
			X	The ES segment was a non-writable segment.
			X	A null ES segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**SUB****Subtract**

Subtracts an immediate value or the value in a register or memory location (second operand) from a register or a memory location (first operand) and stores the result in the first operand location. An immediate value is sign-extended to the length of the first operand.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

The forms of the SUB instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
SUB AL, <i>imm8</i>	2C <i>ib</i>	Subtract an immediate 8-bit value from the AL register and store the result in AL.
SUB AX, <i>imm16</i>	2D <i>iw</i>	Subtract an immediate 16-bit value from the AX register and store the result in AX.
SUB EAX, <i>imm32</i>	2D <i>id</i>	Subtract an immediate 32-bit value from the EAX register and store the result in EAX.
SUB RAX, <i>imm32</i>	2D <i>id</i>	Subtract a sign-extended immediate 32-bit value from the RAX register and store the result in RAX.
SUB <i>reg/mem8, imm8</i>	80 /5 <i>ib</i>	Subtract an immediate 8-bit value from an 8-bit destination register or memory location.
SUB <i>reg/mem16, imm16</i>	81 /5 <i>iw</i>	Subtract an immediate 16-bit value from a 16-bit destination register or memory location.
SUB <i>reg/mem32, imm32</i>	81 /5 <i>id</i>	Subtract an immediate 32-bit value from a 32-bit destination register or memory location.
SUB <i>reg/mem64, imm32</i>	81 /5 <i>id</i>	Subtract a sign-extended immediate 32-bit value from a 64-bit destination register or memory location.
SUB <i>reg/mem16, imm8</i>	83 /5 <i>ib</i>	Subtract a sign-extended immediate 8-bit value from a 16-bit register or memory location.
SUB <i>reg/mem32, imm8</i>	83 /5 <i>ib</i>	Subtract a sign-extended immediate 8-bit value from a 32-bit register or memory location.
SUB <i>reg/mem64, imm8</i>	83 /5 <i>ib</i>	Subtract a sign-extended immediate 8-bit value from a 64-bit register or memory location.
SUB <i>reg/mem8, reg8</i>	28 / <i>r</i>	Subtract the contents of an 8-bit register from an 8-bit destination register or memory location.
SUB <i>reg/mem16, reg16</i>	29 / <i>r</i>	Subtract the contents of a 16-bit register from a 16-bit destination register or memory location.
SUB <i>reg/mem32, reg32</i>	29 / <i>r</i>	Subtract the contents of a 32-bit register from a 32-bit destination register or memory location.
SUB <i>reg/mem64, reg64</i>	29 / <i>r</i>	Subtract the contents of a 64-bit register from a 64-bit destination register or memory location.

Mnemonic	Opcode	Description
SUB reg8, reg/mem8	2A /r	Subtract the contents of an 8-bit register or memory operand from an 8-bit destination register.
SUB reg16, reg/mem16	2B /r	Subtract the contents of a 16-bit register or memory operand from a 16-bit destination register.
SUB reg32, reg/mem32	2B /r	Subtract the contents of a 32-bit register or memory operand from a 32-bit destination register.
SUB reg64, reg/mem64	2B /r	Subtract the contents of a 64-bit register or memory operand from a 64-bit destination register.

## Related Instructions

ADC, ADD, SBB

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## T1MSKC

## Inverse Mask From Trailing Ones

Finds the least significant zero bit in the source operand, clears all bits below that bit to 0, sets all other bits to 1 (including the found bit) and writes the result to the destination. If the least significant bit of the source operand is 0, the destination is written with all ones.

This instruction has two operands:

**T1MSKC** *dest, src*

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The T1MSKC instruction effectively performs a bit-wise logical **or** of the inverse of the source operand and the result of incrementing the source operand by 1 and stores the result to the destination register:

```
add tmp1, src, 1
not tmp2, src
or dest, tmp1, tmp2
```

The value of the carry flag of rFLAGS is generated by the add pseudo-instruction and the remaining arithmetic flags are generated by the or pseudo-instruction.

The T1MSKC instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000\_0001\_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
T1MSKC <i>reg32, reg/mem32</i>	8F	$\overline{\text{RXB}}.09$	0. $\overline{\text{dest}}$ .00	01 /7
T1MSKC <i>reg64, reg/mem64</i>	8F	$\overline{\text{RXB}}.09$	1. $\overline{\text{dest}}$ .00	01 /7

### Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, TZMSK, TZCNT

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
							0				M	M	U	U	M		
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## TEST

## Test Bits

Performs a bit-wise logical and on the value in a register or memory location (first operand) with an immediate value or the value in a register (second operand) and sets the flags in the rFLAGS register based on the result.

This instruction has two operands:

**TEST *dest, src***

While the AND instruction changes the contents of the destination and the flag bits, the TEST instruction changes only the flag bits.

Mnemonic	Opcode	Description
TEST AL, <i>imm8</i>	A8 <i>ib</i>	and an immediate 8-bit value with the contents of the AL register and set rFLAGS to reflect the result.
TEST AX, <i>imm16</i>	A9 <i>iw</i>	and an immediate 16-bit value with the contents of the AX register and set rFLAGS to reflect the result.
TEST EAX, <i>imm32</i>	A9 <i>id</i>	and an immediate 32-bit value with the contents of the EAX register and set rFLAGS to reflect the result.
TEST RAX, <i>imm32</i>	A9 <i>id</i>	and a sign-extended immediate 32-bit value with the contents of the RAX register and set rFLAGS to reflect the result.
TEST <i>reg/mem8, imm8</i>	F6 /0 <i>ib</i>	and an immediate 8-bit value with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem16, imm16</i>	F7 /0 <i>iw</i>	and an immediate 16-bit value with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem32, imm32</i>	F7 /0 <i>id</i>	and an immediate 32-bit value with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem64, imm32</i>	F7 /0 <i>id</i>	and a sign-extended immediate 32-bit value with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem8, reg8</i>	84 /r	and the contents of an 8-bit register with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem16, reg16</i>	85 /r	and the contents of a 16-bit register with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem32, reg32</i>	85 /r	and the contents of a 32-bit register with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result.
TEST <i>reg/mem64, reg64</i>	85 /r	and the contents of a 64-bit register with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result.

## Related Instructions

AND, CMP

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**TZCNT****Count Trailing Zeros**

Counts the number of trailing zero bits in the 16-, 32-, or 64-bit general purpose register or memory source operand. Counting starts upward from the least significant bit and stops when the lowest bit having a value of 1 is encountered or when the most significant bit is encountered. The count is written to the destination register.

If the input operand is zero, CF is set to 1 and the size (in bits) of the input operand is written to the destination register. Otherwise, CF is cleared.

If the least significant bit is a one, the ZF flag is set to 1 and zero is written to the destination register. Otherwise, ZF is cleared.

TZCNT is a BMI instruction. Support for BMI instructions is indicated by CPUID Fn0000\_0007\_EBX\_x0[BMI] = 1. If the TZCNT instruction is not available, the encoding is treated as the BSF instruction. Software *must* check the CPUID bit once per program or library initialization before using the TZCNT instruction or inconsistent behavior may result.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
TZCNT reg16, reg/mem16	F3 0F BC /r	Count the number of trailing zeros in reg/mem16.
TZCNT reg32, reg/mem32	F3 0F BC /r	Count the number of trailing zeros in reg/mem32.
TZCNT reg64, reg/mem64	F3 0F BC /r	Count the number of trailing zeros in reg/mem64.

**Related Instructions**

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZMSK

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

*Note:* Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## TZMSK

## Mask From Trailing Zeros

Finds the least significant one bit in the source operand, sets all bits below that bit to 1, clears all other bits to 0 (including the found bit) and writes the result to the destination. If the least significant bit of the source operand is 1, the destination is written with all zeros.

This instruction has two operands:

**TZMSK *dest*, *src***

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The TZMSK instruction effectively performs a bit-wise logical and of the negation of the source operand and the result of subtracting 1 from the source operand, and stores the result to the destination register:

```
sub tmp1, src, 1
not tmp2, src
and dest, tmp1, tmp2
```

The value of the carry flag of rFLAGS is generated by the sub pseudo-instruction and the remaining arithmetic flags are generated by the and pseudo-instruction.

The TZMSK instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000\_0001\_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
TZMSK reg32, reg/mem32	8F	$\overline{\text{RXB}}.09$	0. $\overline{\text{dest}}$ .0.00	01 /4
TZMSK reg64, reg/mem64	8F	$\overline{\text{RXB}}.09$	1. $\overline{\text{dest}}$ .0.00	01 /4

### Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## UD0, UD1, UD2

## Undefined Operation

These opcodes generate an invalid opcode exception. Unlike other undefined opcodes that may be defined as legal instructions in the future, these opcodes are guaranteed to stay undefined. On some AMD64 processor implementations, UD1 may report an invalid opcode exception regardless of whether fetching the modrm byte could trigger a paging or segmentation exception.

Mnemonic	Opcode	Description
UD0	0F FF	Raise an invalid opcode exception
UD1	0F B9 /r	Raise an invalid opcode exception
UD2	0F 0B	Raise an invalid opcode exception.

### Related Instructions

None

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	This instruction is not recognized.

## WRFSBASE WRGSBASE

## Write FS.base Write GS.base

Writes the base field of the FS or GS segment descriptor with the value contained in the register operand. When supported and enabled, these instructions can be executed at any processor privilege level. Instructions are only defined in 64-bit mode. The address written to the base field must be in canonical form or a #GP fault will occur.

System software must set the FSGSBASE bit (bit 16) of CR4 to enable the WRFSBASE and WRGSBASE instructions.

Support for this instruction is indicated by CPUID Fn0000\_0007\_EBX\_x0[FSGSBASE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 158. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531.

Mnemonic	Opcode	Description
WRFSBASE reg32	F3 0F AE /2	Copy the contents of the specified 32-bit general-purpose register to the lower 32 bits of FS.base.
WRFSBASE reg64	F3 0F AE /2	Copy the contents of the specified 64-bit general-purpose register to FS.base.
WRGSBASE reg32	F3 0F AE /3	Copy the contents of the specified 32-bit general-purpose register to the lower 32 bits of GS.base.
WRGSBASE reg64	F3 0F AE /3	Copy the contents of the specified 64-bit general-purpose register to GS.base.

### Related Instructions

RDFSBASE, RDGSBASE

### rFLAGS Affected

None.

### Exceptions

Exception	Legacy	Compat- ibility	64-bit	Cause of Exception
#UD	X	X		Instruction is not valid in compatibility or legacy modes.
			X	Instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[FSGSBASE] = 0 or, if supported, not enabled in CR4.
#GP			X	Attempt to write non-canonical address to segment base address.

## XADD

## Exchange and Add

Exchanges the contents of a register (second operand) with the contents of a register or memory location (first operand), computes the sum of the two values, and stores the result in the first operand location.

The forms of the XADD instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
XADD reg/mem8, reg8	0F C0 /r	Exchange the contents of an 8-bit register with the contents of an 8-bit destination register or memory operand and load their sum into the destination.
XADD reg/mem16, reg16	0F C1 /r	Exchange the contents of a 16-bit register with the contents of a 16-bit destination register or memory operand and load their sum into the destination.
XADD reg/mem32, reg32	0F C1 /r	Exchange the contents of a 32-bit register with the contents of a 32-bit destination register or memory operand and load their sum into the destination.
XADD reg/mem64, reg64	0F C1 /r	Exchange the contents of a 64-bit register with the contents of a 64-bit destination register or memory operand and load their sum into the destination.

## Related Instructions

None

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**XCHG****Exchange**

Exchanges the contents of the two operands. The operands can be two general-purpose registers or a register and a memory location. If either operand references memory, the processor locks automatically, whether or not the LOCK prefix is used and independently of the value of IOPL. For details about the LOCK prefix, see “Lock Prefix” on page 11.

The x86 architecture commonly uses the XCHG EAX, EAX instruction (opcode 90h) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h as a true NOP only if it would exchange rAX with itself. Without this special handling, the instruction would zero-extend the upper 32 bits of RAX, and thus it would not be a true no-operation. Opcode 90h can still be used to exchange rAX and r8 if the appropriate REX prefix is used.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction.

Mnemonic	Opcode	Description
XCHG AX, <i>reg16</i>	90 +rw	Exchange the contents of the AX register with the contents of a 16-bit register.
XCHG <i>reg16</i> , AX	90 +rw	Exchange the contents of a 16-bit register with the contents of the AX register.
XCHG EAX, <i>reg32</i>	90 +rd	Exchange the contents of the EAX register with the contents of a 32-bit register.
XCHG <i>reg32</i> , EAX	90 +rd	Exchange the contents of a 32-bit register with the contents of the EAX register.
XCHG RAX, <i>reg64</i>	90 +rq	Exchange the contents of the RAX register with the contents of a 64-bit register.
XCHG <i>reg64</i> , RAX	90 +rq	Exchange the contents of a 64-bit register with the contents of the RAX register.
XCHG <i>reg/mem8</i> , <i>reg8</i>	86 /r	Exchange the contents of an 8-bit register with the contents of an 8-bit register or memory operand.
XCHG <i>reg8</i> , <i>reg/mem8</i>	86 /r	Exchange the contents of an 8-bit register or memory operand with the contents of an 8-bit register.
XCHG <i>reg/mem16</i> , <i>reg16</i>	87 /r	Exchange the contents of a 16-bit register with the contents of a 16-bit register or memory operand.
XCHG <i>reg16</i> , <i>reg/mem16</i>	87 /r	Exchange the contents of a 16-bit register or memory operand with the contents of a 16-bit register.
XCHG <i>reg/mem32</i> , <i>reg32</i>	87 /r	Exchange the contents of a 32-bit register with the contents of a 32-bit register or memory operand.
XCHG <i>reg32</i> , <i>reg/mem32</i>	87 /r	Exchange the contents of a 32-bit register or memory operand with the contents of a 32-bit register.
XCHG <i>reg/mem64</i> , <i>reg64</i>	87 /r	Exchange the contents of a 64-bit register with the contents of a 64-bit register or memory operand.
XCHG <i>reg64</i> , <i>reg/mem64</i>	87 /r	Exchange the contents of a 64-bit register or memory operand with the contents of a 64-bit register.

## Related Instructions

BSWAP, XADD

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The source or destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**XLAT****XLATB**

Uses the unsigned integer in the AL register as an offset into a table and copies the contents of the table entry at that location to the AL register.

The instruction uses *seg*:[rBX] as the base address of the table. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix.

This instruction writes AL without changing RAX[63:8]. This instruction ignores operand size.

The single-operand form of the XLAT instruction uses the operand to document the segment and address size attribute, but it uses the base address specified by the rBX register.

This instruction is often used to translate data from one format (such as ASCII) to another (such as EBCDIC).

**Translate Table Index****Mnemonic****Opcode****Description**

XLAT <i>mem8</i>	D7	Set AL to the contents of DS:[rBX + unsigned AL].
XLATB	D7	Set AL to the contents of DS:[rBX + unsigned AL].

**Related Instructions**

None

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

**XOR****Logical Exclusive OR**

Performs a bit-wise logical `xor` operation on both operands and stores the result in the first operand location. The first operand can be a register or memory location. The second operand can be an immediate value, a register, or a memory location. XOR-ing a register with itself clears the register.

The forms of the XOR instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

The instruction performs the following operation for each bit:

<b>X</b>	<b>Y</b>	<b>X xor Y</b>
0	0	0
0	1	1
1	0	1
1	1	0

<b>Mnemonic</b>	<b>Opcode</b>	<b>Description</b>
<code>XOR AL, imm8</code>	34 <i>ib</i>	<code>xor</code> the contents of AL with an immediate 8-bit operand and store the result in AL.
<code>XOR AX, imm16</code>	35 <i>iw</i>	<code>xor</code> the contents of AX with an immediate 16-bit operand and store the result in AX.
<code>XOR EAX, imm32</code>	35 <i>id</i>	<code>xor</code> the contents of EAX with an immediate 32-bit operand and store the result in EAX.
<code>XOR RAX, imm32</code>	35 <i>id</i>	<code>xor</code> the contents of RAX with a sign-extended immediate 32-bit operand and store the result in RAX.
<code>XOR reg/mem8, imm8</code>	80 /6 <i>ib</i>	<code>xor</code> the contents of an 8-bit destination register or memory operand with an 8-bit immediate value and store the result in the destination.
<code>XOR reg/mem16, imm16</code>	81 /6 <i>iw</i>	<code>xor</code> the contents of a 16-bit destination register or memory operand with a 16-bit immediate value and store the result in the destination.
<code>XOR reg/mem32, imm32</code>	81 /6 <i>id</i>	<code>xor</code> the contents of a 32-bit destination register or memory operand with a 32-bit immediate value and store the result in the destination.
<code>XOR reg/mem64, imm32</code>	81 /6 <i>id</i>	<code>xor</code> the contents of a 64-bit destination register or memory operand with a sign-extended 32-bit immediate value and store the result in the destination.
<code>XOR reg/mem16, imm8</code>	83 /6 <i>ib</i>	<code>xor</code> the contents of a 16-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.

Mnemonic	Opcode	Description
XOR reg/mem32, imm8	83 /6 ib	xor the contents of a 32-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.
XOR reg/mem64, imm8	83 /6 ib	xor the contents of a 64-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.
XOR reg/mem8, reg8	30 /r	xor the contents of an 8-bit destination register or memory operand with the contents of an 8-bit register and store the result in the destination.
XOR reg/mem16, reg16	31 /r	xor the contents of a 16-bit destination register or memory operand with the contents of a 16-bit register and store the result in the destination.
XOR reg/mem32, reg32	31 /r	xor the contents of a 32-bit destination register or memory operand with the contents of a 32-bit register and store the result in the destination.
XOR reg/mem64, reg64	31 /r	xor the contents of a 64-bit destination register or memory operand with the contents of a 64-bit register and store the result in the destination.
XOR reg8, reg/mem8	32 /r	xor the contents of an 8-bit destination register with the contents of an 8-bit register or memory operand and store the results in the destination.
XOR reg16, reg/mem16	33 /r	xor the contents of a 16-bit destination register with the contents of a 16-bit register or memory operand and store the results in the destination.
XOR reg32, reg/mem32	33 /r	xor the contents of a 32-bit destination register with the contents of a 32-bit register or memory operand and store the results in the destination.
XOR reg64, reg/mem64	33 /r	xor the contents of a 64-bit destination register with the contents of a 64-bit register or memory operand and store the results in the destination.

## Related Instructions

OR, AND, NOT, NEG

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.



## 4 System Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by the system instructions. System instructions are used to establish the processor operating mode, access processor resources, handle program and system errors, manage memory, and instantiate a virtual machine. Most of these instructions can only be executed by privileged software, such as the operating system or a Virtual Machine Monitor (VMM), also known as a hypervisor. Only system instructions can access certain processor resources, such as the control registers, model-specific registers, and debug registers.

Most system instructions are supported in all hardware implementations of the AMD64 architecture. The table below lists instructions that may not be supported on a given processor implementation. System software must execute the CPUID instruction using the function number listed to determine support prior to using these instructions.

**Table 4-1. System Instruction Support Indicated by CPUID Feature Bits**

Instruction	CPUID Feature Bit	Register[Bit]
Long Mode and Long Mode instructions	CPUID Fn8000_0001_EDX[LM]	EDX[29]
MONITOR, MWAIT	CPUID Fn0000_0001_ECX[MONITOR]	ECX[3]
MONITORX, MWAITX	CPUID CPUID 8000_0001_ECX[MONITORX]	ECX [29]
RDMSR, WRMSR	CPUID Fn0000_0001_EDX[MSR]	EDX[5]
RDTSCP	CPUID Fn8000_0001_EDX[RDTSCP]	EDX[27]
SKINIT, STGI	CPUID Fn8000_0001_ECX[SKINIT]	ECX[12]
SVM Architecture and instructions	CPUID Fn8000_0001_ECX[SVM]	ECX[2]
SYSCALL, SYSRET	CPUID Fn8000_0001_EDX[SysCallSysRet]	EDX[11]
SYSENTER, SYSEXIT	CPUID Fn0000_0001_EDX[SysEnterSysExit]	EDX[11]

There are also several other CPUID feature bits that indicate support for certain paging functions, virtual-mode extensions, machine-check exceptions, advanced programmable interrupt control (APIC), memory-type range registers (MTRRs), etc.

For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158. For a comprehensive list of all instruction support feature flags, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 531. For a comprehensive list of all defined CPUID feature numbers and return values, see Appendix E, “Obtaining Processor Information Via the CPUID Instruction,” on page 601.

For further information about the system instructions and register resources, see:

- “System-Management Instructions” in Volume 2.
- “Summary of Registers and Data Types” on page 38.

- “Notation” on page 52.
- “Instruction Prefixes” on page 5.

**ARPL****Adjust Requestor Privilege Level**

Compares the requestor privilege level (RPL) fields of two segment selectors in the source and destination operands of the instruction. If the RPL field of the destination operand is less than the RPL field of the segment selector in the source register, then the zero flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the destination operand remains unchanged and the zero flag is cleared.

The destination operand can be either a 16-bit register or memory location; the source operand must be a 16-bit register.

The ARPL instruction is intended for use by operating-system procedures 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. The segment selector passed to the operating system is placed in the destination operand and the segment selector for the code segment of the application program is placed in the source operand. The RPL field in the source operand represents the privilege level of the application program. The ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower than the privilege level of the application program.

See “Adjusting Access Rights” in Volume 2, for more information on access rights.

In 64-bit mode, this opcode (63H) is used for the MOVSXD instruction.

Mnemonic	Opcode	Description
ARPL reg/mem16, reg16	63 /r	Adjust the RPL of a destination segment selector to a level not less than the RPL of the segment selector specified in the 16-bit source register. (Invalid in 64-bit mode.)

**Related Instructions**

LAR, LSL, VERR, VERW

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
														M		
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected legacy and compatibility mode.
Stack, #SS			X	A memory address exceeded the stack segment limit.
General protection, #GP			X	A memory address exceeded a data segment limit.
			X	The destination operand was in a non-writable segment.
			X	A null segment selector was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## CLAC

## Clear Alignment Check Flag

Sets the Alignment Check flag in the rFLAGS register to zero. Support for the CLAC instruction is indicated by CPUID Fn07\_EBX[20] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

Mnemonic	Opcode	Description
CLAC	0F 01 CA	Clear AC Flag

### Related Instructions

STAC

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			0													
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
		X	X	Instruction is not supported in virtual mode
	X		X	Lock prefix (F0h) preceding opcode.
			X	CPL was not 0

**CLGI****Clear Global Interrupt Flag**

Clears the global interrupt flag (GIF). While GIF is zero, all external interrupts are disabled.

This is a Secure Virtual Machine instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001(ECX[SVM]=1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume-2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
CLGI	0F 01 DD	Clears the global interrupt flag (GIF).

**Related Instructions**

STGI

**rFLAGS Affected**

None.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001(ECX[SVM]=0.
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.

**CLI****Clear Interrupt Flag**

Clears the interrupt flag (IF) in the rFLAGS register to zero, thereby masking external interrupts received on the INTR input. Interrupts received on the non-maskable interrupt (NMI) input are not affected by this instruction.

In real mode, this instruction clears IF to 0.

In protected mode and virtual-8086-mode, this instruction is IOPL-sensitive. If the CPL is less than or equal to the rFLAGS.IOPL field, the instruction clears IF to 0.

In protected mode, if IOPL < 3, CPL = 3, and protected mode virtual interrupts are enabled (CR4.PVI = 1), then the instruction instead clears rFLAGS.VIF to 0. If none of these conditions apply, the processor raises a general-purpose exception (#GP). For more information, see “Protected Mode Virtual Interrupts” in Volume 2.

In virtual-8086 mode, if IOPL < 3 and the virtual-8086-mode extensions are enabled (CR4.VME = 1), the CLI instruction clears the virtual interrupt flag (rFLAGS.VIF) to 0 instead.

See “Virtual-8086 Mode Extensions” in Volume 2 for more information about IOPL-sensitive instructions.

Mnemonic	Opcode	Description
CLI	FA	Clear the interrupt flag (IF) to zero.

**Action**

```

IF (CPL <= IOPL)
    RFLAGS.IF = 0

ELSEIF (((VIRTUAL_MODE) && (CR4.VME == 1))
        || ((PROTECTED_MODE) && (CR4.PVI == 1) && (CPL == 3)))
    RFLAGS.VIF = 0;

ELSE
    EXCEPTION[#GP(0)]

```

**Related Instructions**

STI

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		M								M						
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		The CPL was greater than the IOPL and virtual mode extensions are not enabled (CR4.VME = 0).
			X	The CPL was greater than the IOPL and either the CPL was not 3 or protected mode virtual interrupts were not enabled (CR4.PVI = 0).

## CLTS

## Clear Task-Switched Flag in CR0

Clears the task-switched (TS) flag in the CR0 register to 0. The processor sets the TS flag on each task switch. The CLTS instruction is intended to facilitate the synchronization of FPU context saves during multitasking operations.

This instruction can only be used if the current privilege level is 0.

See “System-Control Registers” in Volume 2 for more information on FPU synchronization and the TS flag.

Mnemonic	Opcode	Description
CLTS	0F 06	Clear the task-switched (TS) flag in CR0 to 0.

### Related Instructions

LMSW, MOV CR $n$

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

## HLT

## Halt

Causes the microprocessor to halt instruction execution and enter the HALT state. Entering the HALT state puts the processor in low-power mode. Execution resumes when an unmasked hardware interrupt (INTR), non-maskable interrupt (NMI), system management interrupt (SMI), RESET, or INIT occurs.

If an INTR, NMI, or SMI is used to resume execution after a HLT instruction, the saved instruction pointer points to the instruction following the HLT instruction.

Before executing a HLT instruction, hardware interrupts should be enabled. If rFLAGS.IF = 0, the system will remain in a HALT state until an NMI, SMI, RESET, or INIT occurs.

If an SMI brings the processor out of the HALT state, the SMI handler can decide whether to return to the HALT state or not. See *Volume 2: System Programming*, for information on SMIs.

Current privilege level must be 0 to execute this instruction.

Mnemonic	Opcode	Description
HLT	F4	Halt instruction execution.

### Related Instructions

STI, CLI

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

## INT 3

## Interrupt to Debug Vector

Calls the debug exception handler. This instruction maps to a 1-byte opcode (CC) that raises a #BP exception. The INT 3 instruction is normally used by debug software to set instruction breakpoints by replacing the first byte of the instruction opcode bytes with the INT 3 opcode.

This one-byte INT 3 instruction behaves differently from the two-byte INT 3 instruction (opcode CD 03) (see “INT” in Chapter 3 “General Purpose Instructions” for further information) in two ways:

The #BP exception is handled without any IOPL checking in virtual x86 mode. (IOPL mismatches will not trigger an exception.)

- In VME mode, the #BP exception is not redirected via the interrupt redirection table. (Instead, it is handled by a protected mode handler.)

Mnemonic	Opcode	Description
INT 3	CC	Trap to debugger at Interrupt 3.

For complete descriptions of the steps performed by INT instructions, see the following:

- *Legacy-Mode Interrupts*: “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts*: “Long-Mode Interrupt Control Transfers” in Volume 2.

## Action

```
// Refer to INT instruction's Action section for the details on INT_N_REAL,
// INT_N_PROTECTED, and INT_N_VIRTUAL_TO_PROTECTED.
INT3_START:

If (REAL_MODE)
    INT_N_REAL                //N = 3

ELSEIF (PROTECTED_MODE)
    INT_N_PROTECTED           //N = 3

ELSE // VIRTUAL_MODE
    INT_N_VIRTUAL_TO_PROTECTED //N = 3
```

## Related Instructions

INT, INTO, IRET

## rFLAGS Affected

If a task switch occurs, all flags are modified; otherwise, setting are as follows:

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			M	0	0	M				M	0					
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Breakpoint, #BP	X	X	X	INT 3 instruction was executed.
Invalid TSS, #TS (selector)		X	X	As part of a stack switch, the target stack segment selector or rSP in the TSS that was beyond the TSS limit.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was beyond the limit of the GDT or LDT descriptor table.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was a null selector.
		X	X	As part of a stack switch, the target stack segment selector's TI bit was set, but the LDT selector was a null selector.
		X	X	As part of a stack switch, the target stack segment selector in the TSS contained a RPL that was not equal to its DPL.
		X	X	As part of a stack switch, the target stack segment selector in the TSS contained a DPL that was not equal to the CPL of the code segment selector.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was not a writable segment.
Segment not present, #NP (selector)		X	X	The accessed code segment, interrupt gate, trap gate, task gate, or TSS was not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)		X	X	After a stack switch, a memory address exceeded the stack segment limit or was non-canonical and a stack switch occurred.
		X	X	As part of a stack switch, the SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded the data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)	X	X	X	The interrupt vector was beyond the limit of IDT.
		X	X	The descriptor in the IDT was not an interrupt, trap, or task gate in legacy mode or not a 64-bit interrupt or trap gate in long mode.
		X	X	The DPL of the interrupt, trap, or task gate descriptor was less than the CPL.
		X	X	The segment selector specified by the interrupt or trap gate had its TI bit set, but the LDT selector was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate exceeded the descriptor table limit or was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate was not a code segment in legacy mode, or not a 64-bit code segment in long mode.
			X	The DPL of the segment specified by the interrupt or trap gate was greater than the CPL.
		X		The DPL of the segment specified by the interrupt or trap gate pointed was not 0 or it was a conforming segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## INVD

## Invalidate Caches

Invalidates all levels of cache associated with this processor. This may or may not include lower level caches associated with another processor that shares any level of this processor's cache hierarchy.

No data is written back to main memory from invalidating the caches.

CPUID Fn8000\_001D\_EDX[WBINVD]\_xN indicates the behavior of the processor at various levels of the cache hierarchy. If the feature bit is 0, the instruction causes the invalidation of all lower level caches of other processors sharing the designated level of cache. If the feature bit is 1, the instruction does not necessarily cause the invalidation of all lower level caches of other processors sharing the designated level of cache. See Appendix E, “Obtaining Processor Information Via the CPUID Instruction,” on page 601 for more information on using the CPUID function.

This is a privileged instruction. The current privilege level (CPL) of a procedure invalidating the processor's internal caches must be 0.

To insure that data is written back to memory prior to invalidating caches, use the WBINVD instruction.

This instruction does not invalidate TLB caches.

INVD is a serializing instruction.

Mnemonic	Opcode	Description
INVD	OF 08	Invalidate internal caches and trigger external cache invalidations.

## Related Instructions

WBINVD, CLFLUSH

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

**INVLPG****Invalidate TLB Entry**

Invalidates the TLB entry that would be used for the 1-byte memory operand.

This instruction invalidates the TLB entry, regardless of the G (Global) bit setting in the associated PDE or PTE entry and regardless of the page size (4 Kbytes, 2 Mbytes, 4 Mbytes, or 1 Gbyte). It may invalidate any number of additional TLB entries, in addition to the targeted entry.

INVLPG is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction.

See “Page Translation and Protection” in Volume 2 for more information on page translation.

Mnemonic	Opcode	Description
INVLPG <i>mem8</i>	0F 01 /7	Invalidate the TLB entry for the page containing a specified memory location.

**Related Instructions**

INVLPGA, MOV CR*n* (CR3 and CR4)

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

**INVLPGA****Invalidate TLB Entry in a Specified ASID**

Invalidates the TLB mapping for a given virtual page and a given ASID. The virtual (linear) address is specified in the implicit register operand rAX. The portion of RAX used to form the address is determined by the effective address size (current execution mode and optional address size prefix). The ASID is taken from ECX.

The INVLPGA instruction may invalidate any number of additional TLB entries, in addition to the targeted entry.

The INVLPGA instruction is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001(ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume-2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
INVLPGA rAX, ECX	0F 01 DF	Invalidates the TLB mapping for the virtual page specified in rAX and the ASID specified in ECX.

**Related Instructions**

INVLPG.

**rFLAGS Affected**

None.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001(ECX[SVM] = 0).
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.

**IRET****IRETD****IRETQ****Return from Interrupt**

Returns program control from an exception or interrupt handler to a program or procedure previously interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions also perform a return from a nested task. All flags, CS, and rIP are restored to the values they had before the interrupt so that execution may continue at the next instruction following the interrupt or exception. In 64-bit mode or if the CPL changes, SS and RSP are also restored.

IRET, IRETD, and IRETQ are synonyms mapping to the same opcode. They are intended to provide semantically distinct forms for various opcode sizes. The IRET instruction is used for 16-bit operand size; IRETD is used for 32-bit operand sizes; IRETQ is used for 64-bit operands. The latter form is only meaningful in 64-bit mode.

IRET, IRETD, or IRETQ must be used to terminate the exception or interrupt handler associated with the exception, external interrupt, or software-generated interrupt.

IRET<sub>x</sub> is a serializing instruction.

For detailed descriptions of the steps performed by IRET<sub>x</sub> instructions, see the following:

- *Legacy-Mode Interrupts*: “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts*: “Long-Mode Interrupt Control Transfers” in Volume 2.

Mnemonic	Opcode	Description
IRET	CF	Return from interrupt (16-bit operand size).
IRETD	CF	Return from interrupt (32-bit operand size).
IRETQ	CF	Return from interrupt (64-bit operand size).

**Action**

IRET\_START:

```
IF (REAL_MODE)
    IRET_REAL
ELSIF (PROTECTED_MODE)
    IRET_PROTECTED
ELSE // (VIRTUAL_MODE)
    IRET_VIRTUAL
```

IRET\_REAL:

```
POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS
```

```
IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(0)]  
  
CS.sel = temp_CS
CS.base = temp_CS SHL 4  
  
RFLAGS.v = temp_RFLAGS // VIF,VIP,VM unchanged
RIP = temp_RIP
EXIT  
  
IRET_PROTECTED:  
  
IF (RFLAGS.NT==1)           // iret does a task-switch to a previous task
    IF (LEGACY_MODE)
        TASK_SWITCH          // using the 'back link' field in the tss
    ELSE
        // (LONG_MODE)
        EXCEPTION [#GP(0)] // task switches aren't supported in long mode  
  
POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS  
  
IF ((temp_RFLAGS.VM==1) && (CPL==0) && (LEGACY_MODE))
    IRET_FROM_PROTECTED_TO_VIRTUAL  
  
temp_CPL = temp_CS.rpl  
  
IF ((64BIT_MODE) || (temp_CPL!=CPL))
{
    POP.v temp_RSP          // in 64-bit mode, iret always pops ss:rsp
    POP.v temp_SS
}  
  
CS = READ_DESCRIPTOR (temp_CS, iret_chk)  
  
IF ((64BIT_MODE) && (temp_RIP is non-canonical)
    || (!64BIT_MODE) && (temp_RIP > CS.limit))
{
    EXCEPTION [#GP(0)]
}  
  
CPL = temp_CPL  
  
IF ((started in 64-bit mode) || (changing CPL))
    // ss:rsp were popped, so load them into the registers
{
    SS = READ_DESCRIPTOR (temp_SS, ss_chk)
    RSP.s = temp_RSP
}
```

```

IF (changing CPL)
{
    FOR (seg = ES, DS, FS, GS)
        IF ((seg.attr.dpl < CPL) && ((seg.attr.type == 'data')
            || (seg.attr.type == 'non-conforming-code')))
        {
            seg = NULL           // can't use lower dpl data segment at higher cpl
        }
    RFLAGS.v = temp_RFLAGS      // VIF,VIP,IOPL only changed if (old_CPL==0)
                                // IF only changed if (old_CPL<=old_RFLAGS.IOPL)
                                // VM unchanged
                                // RF cleared
    RIP = temp_RIP
    EXIT
}

IRET_VIRTUAL:

IF ((RFLAGS.IOPL<3) && (CR4.VME==0))
    EXCEPTION [#GP(0)]

POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS

IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(0)]

IF (RFLAGS.IOPL==3)
{
    RFLAGS.v = temp_RFLAGS // VIF,VIP,VM,IOPL unchanged
                            // RF cleared
    CS.sel = temp_CS
    CS.base = temp_CS SHL 4

    RIP = temp_RIP
    EXIT
}

// now ((IOPL<3) && (CR4.VME==1))

ELSIF ((OPERAND_SIZE==16)
    && !((temp_RFLAGS.IF==1) && (RFLAGS.VIP==1))
    && (temp_RFLAGS.TF==0))
{
    RFLAGS.w = temp_RFLAGS // RFLAGS.VIF=temp_RFLAGS.IF
                            // IF,IOPL unchanged
                            // RF cleared
    CS.sel = temp_CS
    CS.base = temp_CS SHL 4
}

```

```
RIP = temp_RIP
EXIT
}
ELSE // ((RFLAGS.IOPL<3) && (CR4.VME==1) && ((OPERAND_SIZE==32) ||
// ((temp_RFLAGS.IF==1) && (RFLAGS.VIP==1)) || (temp_RFLAGS.TF==1)))
EXCEPTION [#GP(0)]
```

IRET\_FROM\_PROTECTED\_TO\_VIRTUAL:

```
// temp_RIP already popped
// temp_CS already popped
// temp_RFLAGS already popped, temp_RFLAGS.VM=1

POP.d temp_RSP
POP.d temp_SS
POP.d temp_ES
POP.d temp_DS
POP.d temp_FS
POP.d temp_GS

CS.sel = temp_CS           // force the segments to have virtual-mode values
CS.base = temp_CS SHL 4
CS.limit= 0x0000FFFF
CS.attr = 16-bit dpl3 code

SS.sel = temp_SS
SS.base = temp_SS SHL 4
SS.limit= 0x0000FFFF
SS.attr = 16-bit dpl3 stack

DS.sel = temp_DS
DS.base = temp_DS SHL 4
DS.limit= 0x0000FFFF
DS.attr = 16-bit dpl3 data

ES.sel = temp_ES
ES.base = temp_ES SHL 4
ES.limit= 0x0000FFFF
ES.attr = 16-bit dpl3 data

FS.sel = temp_FS
FS.base = temp_FS SHL 4
FS.limit= 0x0000FFFF
FS.attr = 16-bit dpl3 data

GS.sel = temp_GS
GS.base = temp_GS SHL 4
GS.limit= 0x0000FFFF
GS.attr = 16-bit dpl3 data
```

```

RSP.d = temp_RSP
RFLAGS.d = temp_RFLAGS
CPL = 3

RIP = temp_RIP AND 0x0000FFFF
EXIT

```

## Related Instructions

INT, INTO, INT3

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Segment not present, #NP (selector)			X	The return code segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
		X		IOPL was less than 3 and one of the following conditions was true: <ul style="list-style-type: none"><li>• CR4.VME was 0.</li><li>• The effective operand size was 32-bit.</li><li>• Both the original EFLAGS.VIP and the new EFLAGS.IF were set.</li><li>• The new EFLAGS.TF was set.</li></ul>
			X	IRET <sub>x</sub> was executed in long mode while EFLAGS.NT=1.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)			X	The return code selector was a null selector.
			X	The return stack selector was a null selector and the return mode was non-64-bit mode or CPL was 3.
			X	The return code or stack descriptor exceeded the descriptor table limit.
			X	The return code or stack selector's TI bit was set but the LDT selector was a null selector.
			X	The segment descriptor for the return code was not a code segment.
			X	The RPL of the return code segment selector was less than the CPL.
			X	The return code segment was non-conforming and the segment selector's DPL was not equal to the RPL of the code segment's segment selector.
			X	The return code segment was conforming and the segment selector's DPL was greater than the RPL of the code segment's segment selector.
			X	The segment descriptor for the return stack was not a writable data segment.
			X	The stack segment descriptor DPL was not equal to the RPL of the return code segment selector.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**LAR****Load Access Rights Byte**

Loads the access rights from the segment descriptor specified by a 16-bit source register or memory operand into a specified 16-bit, 32-bit, or 64-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful. LAR clears the zero flag if the descriptor is invalid for any reason.

The LAR instruction checks that:

- the segment selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the descriptor DPL is greater than or equal to both the CPL and RPL, or the segment is a conforming code segment.
- the descriptor type is valid for the LAR instruction. Valid descriptor types are shown in the following table. LDT and TSS descriptors in 64-bit mode, and call-gate descriptors in long mode, are only valid if bits 12:8 of doubleword +12 are zero.

See Volume 2, Section 6.4 for more information on checking access rights using LAR.

Valid Descriptor Type		Description
Legacy Mode	Long Mode	
All	All	All code and data descriptors
1	—	Available 16-bit TSS
2	2	LDT
3	—	Busy 16-bit TSS
4	—	16-bit call gate
5	—	Task gate
9	9	Available 32-bit or 64-bit TSS
B	B	Busy 32-bit or 64-bit TSS
C	C	32-bit or 64-bit call gate

If the segment descriptor passes these checks, the attributes are loaded into the destination general-purpose register. If it does not, then the zero flag is cleared and the destination register is not modified.

When the operand size is 16 bits, access rights include the DPL and Type fields located in bytes 4 and 5 of the descriptor table entry. Before loading the access rights into the destination operand, the low order word is masked with FF00H.

When the operand size is 32 or 64 bits, access rights include the DPL and type as well as the descriptor type (S field), segment present (P flag), available to system (AVL flag), default operation size (D/B

flag), and granularity flags located in bytes 4–7 of the descriptor. Before being loaded into the destination operand, the doubleword is masked with 00FF\_FF00H.

In 64-bit mode, for both 32-bit and 64-bit operand sizes, 32-bit register results are zero-extended to 64 bits.

This instruction can only be executed in protected mode.

Mnemonic	Opcode	Description
LAR reg16, reg/mem16	0F 02 /r	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with FF00h and saves the result in the 16-bit destination register.
LAR reg32, reg/mem16	0F 02 /r	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 32-bit destination register.
LAR reg64, reg/mem16	0F 02 /r	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 64-bit destination register.

## Related Instructions

ARPL, LSL, VERR, VERW

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
														M		
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded the data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**LGDT****Load Global Descriptor Table Register**

Loads the pseudo-descriptor specified by the source operand into the global descriptor table register (GDTR). The pseudo-descriptor is a memory location containing the GDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is 6 bytes; in 64-bit mode, it is 10 bytes.

If the operand size is 16 bits, the high-order byte of the 6-byte pseudo-descriptor is not used. The lower two bytes specify the 16-bit limit and the third, fourth, and fifth bytes specify the 24-bit base address. The high-order byte of the GDTR is filled with zeros.

If the operand size is 32 bits, the lower two bytes specify the 16-bit limit and the upper four bytes specify a 32-bit base address.

In 64-bit mode, the lower two bytes specify the 16-bit limit and the upper eight bytes specify a 64-bit base address. In 64-bit mode, operand-size prefixes are ignored and the operand size is forced to 64-bits; therefore, the pseudo-descriptor is always 10 bytes.

This instruction is only used in operating system software and must be executed at CPL 0. It is typically executed once in real mode to initialize the processor before switching to protected mode.

LGDT is a serializing instruction.

Mnemonic	Opcode	Description
LGDT <i>mem16:32</i>	0F 01 /2	Loads <i>mem16:32</i> into the global descriptor table register.
LGDT <i>mem16:64</i>	0F 01 /2	Loads <i>mem16:64</i> into the global descriptor table register.

**Related Instructions**

LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X		X	A memory address exceeded the stack segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X		X	A memory address exceeded the data segment limit or was non-canonical.
		X	X	CPL was not 0.
			X	The new GDT base address was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

**LIDT****Load Interrupt Descriptor Table Register**

Loads the pseudo-descriptor specified by the source operand into the interrupt descriptor table register (IDTR). The pseudo-descriptor is a memory location containing the IDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is six bytes; in 64-bit mode, it is 10 bytes.

If the operand size is 16 bits, the high-order byte of the 6-byte pseudo-descriptor is not used. The lower two bytes specify the 16-bit limit and the third, fourth, and fifth bytes specify the 24-bit base address. The high-order byte of the IDTR is filled with zeros.

If the operand size is 32 bits, the lower two bytes specify the 16-bit limit and the upper four bytes specify a 32-bit base address.

In 64-bit mode, the lower two bytes specify the 16-bit limit, and the upper eight bytes specify a 64-bit base address. In 64-bit mode, operand-size prefixes are ignored and the operand size is forced to 64-bits; therefore, the pseudo-descriptor is always 10 bytes.

This instruction is only used in operating system software and must be executed at CPL 0. It is normally executed once in real mode to initialize the processor before switching to protected mode.

LIDT is a serializing instruction.

Mnemonic	Opcode	Description
LIDT <i>mem16:32</i>	0F 01 /3	Loads <i>mem16:32</i> into the interrupt descriptor table register.
LIDT <i>mem16:64</i>	0F 01 /3	Loads <i>mem16:64</i> into the interrupt descriptor table register.

**Related Instructions**

LGDT, LLDT, LTR, SGDT, SIDT, SLDT, STR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X		X	A memory address exceeded the stack segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X		X	A memory address exceeded the data segment limit or was non-canonical.
		X	X	CPL was not 0.
			X	The new IDT base address was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

**LLDT****Load Local Descriptor Table Register**

Loads the specified segment selector into the visible portion of the local descriptor table (LDT). The processor uses the selector to locate the descriptor for the LDT in the global descriptor table. It then loads this descriptor into the hidden portion of the LDTR.

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

In legacy and compatibility modes, the LDT descriptor is 8 bytes long and contains a 32-bit base address.

In 64-bit mode, the LDT descriptor is 16-bytes long and contains a 64-bit base address. The LDT descriptor type (02h) is redefined in 64-bit mode for use as the 16-byte LDT descriptor.

This instruction must be executed in protected mode. It is only provided for use by operating system software at CPL 0.

LLDT is a serializing instruction.

Mnemonic	Opcode	Description
LLDT reg/mem16	0F 00 /2	Load the 16-bit segment selector into the local descriptor table register and load the LDT descriptor from the GDT.

**Related Instructions**

LGDT, LIDT, LTR, SGDT, SIDT, SLDT, STR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Segment not present, #NP (selector)			X	The LDT descriptor was marked not present.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	CPL was not 0.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)			X	The source selector did not point into the GDT.
			X	The descriptor was beyond the GDT limit.
			X	The descriptor was not an LDT descriptor.
			X	The descriptor's extended attribute bits were not zero in 64-bit mode.
			X	The new LDT base address was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

## LMSW

## Load Machine Status Word

Loads the lower four bits of the 16-bit register or memory operand into bits 3:0 of the machine status word in register CR0. Only the protection enabled (PE), monitor coprocessor (MP), emulation (EM), and task switched (TS) bits of CR0 are modified. Additionally, LMSW can set CR0.PE, but cannot clear it.

The LMSW instruction can be used only when the current privilege level is 0. It is only provided for compatibility with early processors.

Use the MOV CR0 instruction to load all 32 or 64 bits of CR0.

Mnemonic	Opcode	Description
LMSW <i>reg/mem16</i>	0F 01 /6	Load the lower 4 bits of the source into the lower 4 bits of CR0.

### Related Instructions

MOV CR*n*, SMSW

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X		X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X		X	A memory address exceeded a data segment limit or was non-canonical.
		X	X	CPL was not 0.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

**LSL****Load Segment Limit**

Loads the segment limit from the segment descriptor specified by a 16-bit source register or memory operand into a specified 16-bit, 32-bit, or 64-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful. LSL clears the zero flag if the descriptor is invalid for any reason.

In 64-bit mode, for both 32-bit and 64-bit operand sizes, 32-bit register results are zero-extended to 64 bits.

The LSL instruction checks that:

- the segment selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the descriptor DPL is greater than or equal to both the CPL and RPL, or the segment is a conforming code segment.
- the descriptor type is valid for the LAR instruction. Valid descriptor types are shown in the following table. LDT and TSS descriptors in 64-bit mode are only valid if bits 12:8 of doubleword +12 are zero, as described in “System Descriptors” in Volume 2.

Valid Descriptor Type		Description
Legacy Mode	Long Mode	
—	—	All code and data descriptors
1	—	Available 16-bit TSS
2	2	LDT
3	—	Busy 16-bit TSS
9	9	Available 32-bit or 64-bit TSS
B	B	Busy 32-bit or 64-bit TSS

If the segment selector passes these checks and the segment limit is loaded into the destination general-purpose register, the instruction sets the zero flag of the rFLAGS register to 1. If the selector does not pass the checks, then LSL clears the zero flag to 0 and does not modify the destination.

The instruction calculates the segment limit to 32 bits, taking the 20-bit limit and the granularity bit into account. When the operand size is 16 bits, it truncates the upper 16 bits of the 32-bit adjusted segment limit and loads the lower 16-bits into the target register.

Mnemonic	Opcode	Description
LSL reg16, reg/mem16	0F 03 /r	Loads a 16-bit general-purpose register with the segment limit for a selector specified in a 16-bit memory or register operand.

LSL reg32, reg/mem16 0F 03 /r

Loads a 32-bit general-purpose register with the segment limit for a selector specified in a 16-bit memory or register operand.

LSL reg64, reg/mem16 0F 03 /r

Loads a 64-bit general-purpose register with the segment limit for a selector specified in a 16-bit memory or register operand.

## Related Instructions

ARPL, LAR, VERR, VERW

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**LTR****Load Task Register**

Loads the specified segment selector into the visible portion of the task register (TR). The processor uses the selector to locate the descriptor for the TSS in the global descriptor table. It then loads this descriptor into the hidden portion of TR. The TSS descriptor in the GDT is marked busy, but no task switch is made.

If the source operand is null, a general protection exception (#GP) is generated.

In legacy and compatibility modes, the TSS descriptor is 8 bytes long and contains a 32-bit base address.

In 64-bit mode, the instruction references a 64-bit descriptor to load a 64-bit base address. The TSS type (09H) is redefined in 64-bit mode for use as the 16-byte TSS descriptor.

This instruction must be executed in protected mode when the current privilege level is 0. It is only provided for use by operating system software.

The operand size attribute has no effect on this instruction.

LTR is a serializing instruction.

Mnemonic	Opcode	Description
LTR reg/mem16	0F 00 /3	Load the 16-bit segment selector into the task register and load the TSS descriptor from the GDT.

**Related Instructions**

LGDT, LIDT, LLDT, STR, SGDT, SIDT, SLDT

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Segment not present, #NP (selector)			X	The TSS descriptor was marked not present.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	CPL was not 0.
			X	A null data segment was used to reference memory.
			X	The new TSS selector was a null selector.
General protection, #GP (selector)			X	The source selector did not point into the GDT.
			X	The descriptor was beyond the GDT limit.
			X	The descriptor was not an available TSS descriptor.
			X	The descriptor's extended attribute bits were not zero in 64-bit mode.
			X	The new TSS base address was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

## MONITOR

## Setup Monitor Address

Establishes a linear address range of memory for hardware to monitor and puts the processor in the monitor event pending state. When in the monitor event pending state, the monitoring hardware detects stores to the specified linear address range and causes the processor to exit the monitor event pending state. The MWAIT instruction uses the state of the monitor hardware.

The address range should be a write-back memory type. Executing MONITOR on an address range for a non-write-back memory type is not guaranteed to cause the processor to enter the monitor event pending state. The size of the linear address range that is established by the MONITOR instruction can be determined by CPUID function 0000\_0005h.

The [rAX] register provides the effective address. The DS segment is the default segment used to create the linear address. Segment overrides may be used with the MONITOR instruction.

The ECX register specifies optional extensions for the MONITOR instruction. There are currently no extensions defined and setting any bits in ECX will result in a #GP exception. The ECX register operand is implicitly 32-bits.

The EDX register specifies optional hints for the MONITOR instruction. There are currently no hints defined and EDX is ignored by the processor. The EDX register operand is implicitly 32-bits.

The MONITOR instruction can be executed at CPL 0 and is allowed at CPL > 0 only if MSR C001\_0015h[MonMwaitUserEn] = 1. When MSR C001\_0015h[MonMwaitUserEn] = 0, MONITOR generates #UD at CPL > 0. (See the *BIOS and Kernel Developer's Guide* applicable to your product for specific details on MSR C001\_0015h.)

MONITOR performs the same segmentation and paging checks as a 1-byte read.

Support for the MONITOR instruction is indicated by CPUID Fn0000\_0001\_ECX[MONITOR] = 1. Software must check the CPUID bit once per program or library initialization before using the MONITOR instruction, or inconsistent behavior may result. Software designed to run at CPL greater than 0 must also check for availability by testing whether executing MONITOR causes a #UD exception.

The following pseudo-code shows typical usage of a MONITOR/MWAIT pair:

```
EAX = Linear_Address_to_Monitor;
ECX = 0; // Extensions
EDX = 0; // Hints

while (!matching_store_done) {
    MONITOR EAX, ECX, EDX
    IF (!matching_store_done) {
        MWAIT EAX, ECX
    }
}
```

Mnemonic	Opcode	Description
MONITOR	0F 01 C8	Establishes a linear address range to be monitored by hardware and activates the monitor hardware.

## Related Instructions

MWAIT, MONITORX, MWAITX

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The MONITOR/MWAIT instructions are not supported, as indicated by CPUID Fn0000_0001_ECX[MONITOR] = 0.
		X	X	CPL was not zero and MSR C001_0015[MonMwaitUserEn] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	ECX was non-zero.
			X	A null data segment was used to reference memory.
Page Fault, #PF		X	X	A page fault resulted from the execution of the instruction.

## MOV CR $n$

## Move to/from Control Registers

Moves the contents of a 32-bit or 64-bit general-purpose register to a control register or vice versa.

In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix. In non-64-bit mode, the operand size is fixed at 32 bits and the upper 32 bits of the destination are forced to 0.

CR0 maintains the state of various control bits. CR2 and CR3 are used for page translation. CR4 holds various feature enable bits. CR8 is used to prioritize external interrupts. CR1, CR5, CR6, CR7, and CR9 through CR15 are all reserved and raise an undefined opcode exception (#UD) if referenced.

CR8 can be read and written in 64-bit mode, using a REX prefix. CR8 can be read and written in all modes using a LOCK prefix instead of a REX prefix to specify the additional opcode bit. To verify whether the LOCK prefix can be used in this way, check for support of this feature. CPUID Fn8000\_0001\_ECX[AltMovCr8] = 1, indicates that this feature is supported.

For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

CR8 can also be read and modified using the task priority register described in “System-Control Registers” in Volume 2.

This instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

MOV CR $n$  is a privileged instruction and must always be executed at CPL = 0.

MOV CR $n$  is a serializing instruction.

Mnemonic	Opcode	Description
MOV CR $n$ , reg32	0F 22 /r	Move the contents of a 32-bit register to CR $n$
MOV CR $n$ , reg64	0F 22 /r	Move the contents of a 64-bit register to CR $n$
MOV reg32, CR $n$	0F 20 /r	Move the contents of CR $n$ to a 32-bit register.
MOV reg64, CR $n$	0F 20 /r	Move the contents of CR $n$ to a 64-bit register.
MOV CR8, reg32	F0 0F 22/r	Move the contents of a 32-bit register to CR8.
MOV CR8, reg64	F0 0F 22/r	Move the contents of a 64-bit register to CR8.
MOV reg32, CR8	F0 0F 20/r	Move the contents of CR8 into a 32-bit register.
MOV reg64, CR8	F0 0F 20/r	Move the contents of CR8 into a 64-bit register.

### Related Instructions

CLTS, LMSW, SMSW

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid Instruction, #UD	X	X	X	An illegal control register was referenced (CR1, CR5–CR7, CR9–CR15).
	X	X	X	The use of the LOCK prefix to read CR8 is not supported, as indicated by CPUID Fn8000_0001_ECX[AltMovCr8] = 0.
General protection, #GP		X	X	CPL was not 0.
	X		X	An attempt was made to set CR0.PG = 1 and CR0.PE = 0.
	X		X	An attempt was made to set CR0.CD = 0 and CR0.NW = 1.
	X		X	Reserved bits were set in the page-directory pointers table (used in the legacy extended physical addressing mode) and the instruction modified CR0, CR3, or CR4.
	X		X	An attempt was made to write 1 to any reserved bit in CR0, CR3, CR4 or CR8.
	X		X	An attempt was made to set CR0.PG while long mode was enabled (EFER.LME = 1), but paging address extensions were disabled (CR4.PAE = 0).
			X	An attempt was made to clear CR4.PAE while long mode was active (EFER.LMA = 1).

## MOV DR $n$

## Move to/from Debug Registers

Moves the contents of a debug register into a 32-bit or 64-bit general-purpose register or vice versa.

In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix. In non-64-bit mode, the operand size is fixed at 32-bits and the upper 32 bits of the destination are forced to 0.

DR0 through DR3 are linear breakpoint address registers. DR6 is the debug status register and DR7 is the debug control register. DR4 and DR5 are aliased to DR6 and DR7 if CR4.DE = 0, and are reserved if CR4.DE = 1.

DR8 through DR15 are reserved and generate an undefined opcode exception if referenced.

These instructions are privileged and must be executed at CPL 0.

The MOV DR $n$ , reg32 and MOV DR $n$ , reg64 instructions are serializing instructions.

The MOV(DR) instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

See “Debug and Performance Resources” in Volume 2 for details.

Mnemonic	Opcode	Description
MOV reg32, DR $n$	0F 21 /r	Move the contents of DR $n$ to a 32-bit register.
MOV reg64, DR $n$	0F 21 /r	Move the contents of DR $n$ to a 64-bit register.
MOV DR $n$ , reg32	0F 23 /r	Move the contents of a 32-bit register to DR $n$ .
MOV DR $n$ , reg64	0F 23 /r	Move the contents of a 64-bit register to DR $n$ .

### Related Instructions

None

### rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Debug, #DB	X		X	A debug register was referenced while the general detect (GD) bit in DR7 was set.
Invalid opcode, #UD	X		X	DR4 or DR5 was referenced while the debug extensions (DE) bit in CR4 was set.
			X	An illegal debug register (DR8–DR15) was referenced.
General protection, #GP		X	X	CPL was not 0.
			X	A 1 was written to any of the upper 32 bits of DR6 or DR7 in 64-bit mode.

## MWAIT

## Monitor Wait

Used in conjunction with the MONITOR instruction to cause a processor to wait until a store occurs to a specific linear address range from another processor. The previously executed MONITOR instruction causes the processor to enter the monitor event pending state. The MWAIT instruction may enter an implementation dependent power state until the monitor event pending state is exited. The MWAIT instruction has the same effect on architectural state as the NOP instruction.

Events that cause an exit from the monitor event pending state include:

- A store from another processor matches the address range established by the MONITOR instruction.
- Any unmasked interrupt, including INTR, NMI, SMI, INIT.
- RESET.
- Any far control transfer that occurs between the MONITOR and the MWAIT.

EAX specifies optional hints for the MWAIT instruction. Optimized C-state request is communicated through EAX[7:4]. The processor C-state is EAX[7:4]+1, so to request C0 is to place the value F in EAX[7:4] and to request C1 is to place the value 0 in EAX[7:4]. All other components of EAX should be zero when making the C1 request. Setting a reserved bit in EAX is ignored by the processor.

ECX specifies optional extensions for the MWAIT instruction. The only extension currently defined is ECX bit 0, which allows interrupts to wake MWAIT, even when eFLAGS.IF = 0. Support for this extension is indicated by a feature flag returned by the CPUID instruction. Setting any unsupported bit in ECX results in a #GP exception.

CPUID Function 0000\_0005h indicates support for extended features of MONITOR/MWAIT:

- CPUID Fn0000\_0005\_ECX[EMX] = 1 indicates support for enumeration of MONITOR/MWAIT extensions.
- CPUID Fn0000\_0005\_ECX[IBE] = 1 indicates that MWAIT can set ECX[0] to allow interrupts to cause an exit from the monitor event pending state even when eFLAGS.IF = 0.

The MWAIT instruction can be executed at CPL 0 and is allowed at CPL > 0 only if MSR C001\_0015h[MonMwaitUserEn] = 1. When MSR C001\_0015h[MonMwaitUserEn] is 0, MWAIT generates #UD at CPL > 0. (See the *BIOS and Kernel Developer's Guide* applicable to your product for specific details on MSR C001\_0015h.)

Support for the MWAIT instruction is indicated by CPUID Fn0000\_0001\_ECX[MONITOR] = 1. Software MUST check the CPUID bit once per program or library initialization before using the MWAIT instruction, or inconsistent behavior may result. Software designed to run at CPL greater than 0 must also check for availability by testing whether executing MWAIT causes a #UD exception.

The use of the MWAIT instruction is contingent upon the satisfaction of the following coding requirements:

- MONITOR must precede the MWAIT and occur in the same loop.

- MWAIT must be conditionally executed only if the awaited store has not already occurred. (This prevents a race condition between the MONITOR instruction arming the monitoring hardware and the store intended to trigger the monitoring hardware.)

The following pseudo-code shows typical usage of a MONITOR/MWAIT pair:

```
EAX = Linear_Address_to_Monitor;
ECX = 0; // Extensions
EDX = 0; // Hints

WHILE (!matching_store_done){
    MONITOR EAX, ECX, EDX
    IF ( !matching_store_done ) {
        MWAIT EAX, ECX
    }
}
```

Mnemonic	Opcode	Description
MWAIT	0F 01 C9	Causes the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

## Related Instructions

MONITOR

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The MONITOR/MWAIT instructions are not supported, as indicated by CPUID Fn0000_0001(ECX)[MONITOR] = 0.
		X	X	CPL was not zero and MSRC001_0015[MonMwaitUserEn] = 0.
General protection, #GP	X	X	X	Unsupported extension bits were set in ECX

## RDMSR

## Read Model-Specific Register

Loads the contents of a 64-bit model-specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register receives the high-order 32 bits and the EAX register receives the low order bits. The RDMSR instruction ignores operand size; ECX always holds the MSR number, and EDX:EAX holds the data. If a model-specific register has fewer than 64 bits, the unimplemented bit positions loaded into the destination registers are undefined.

This instruction must be executed at a privilege level of 0 or a general protection exception (#GP) will be raised. This exception is also generated if a reserved or unimplemented model-specific register is specified in ECX.

Support for the RDMSR instruction is indicated by CPUID Fn0000\_0001\_EDX[MSR] = 1 OR CPUID Fn8000\_0001\_EDX[MSR] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

For more information about model-specific registers, see the documentation for various hardware implementations and “Model-Specific Registers (MSRs)” in *Volume 2: System Programming*.

Mnemonic	Opcode	Description
RDMSR	0F 32	Copy MSR specified by ECX into EDX:EAX.

### Related Instructions

WRMSR, RDTSC, RDPMC

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The RDMSR instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[MSR] = 0 or CPUID Fn8000_0001_EDX[MSR] = 0.
General protection, #GP		X	X	CPL was not 0.
	X		X	The value in ECX specifies a reserved or unimplemented MSR address.

## RDPMC

## Read Performance-Monitoring Counter

Reads the contents of a 64-bit performance counter and returns it in the registers EDX:EAX. The ECX register is used to specified by the index of the performance counter to be read. The EDX register receives the high-order 32 bits and the EAX register receives the low order 32 bits of the counter. The RDPMC instruction ignores operand size; the index and the return values are all 32 bits.

The base architecture supports four core performance counters: PerfCtr0–3. An extension to the architecture increases the number of core performance counters to 6 (PerfCtr0–5). Other extensions add four northbridge performance counters NB\_PerfCtr0–3 and four L2 cache performance counters L2I\_PerfCtr0–3.

To select the core performance counter to be read, specify the counter index, rather than the performance counter MSR address. To access the northbridge performance counters, specify the index of the counter plus 6. To access the L2 cache performance counters, specify the index of the counter plus 10.

Programs running at any privilege level can read performance monitor counters if the PCE flag in CR4 is set to 1; otherwise this instruction must be executed at a privilege level of 0.

This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the performance counter is read.

For more information about performance-counter registers, see the documentation for various hardware implementations and “Performance Counters” in Volume 2.

Support for the core performance counters PerfCtr4–5 is indicated by CPUID Fn8000\_0001\_ECX[PerfCtrExtCore] = 1. CPUID Fn8000\_0001\_ECX[PerfCtrExtNB] = 1 indicates support for the four architecturally defined northbridge performance counters and CPUID Fn8000\_0001\_ECX[PerfCtrExtL2I] = 1 indicates support for the L2 cache performance counters.

For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

### Instruction Encoding

Mnemonic	Opcode	Description
RDPMC	0F 33	Copy the performance monitor counter specified by ECX into EDX:EAX.

### Related Instructions

RDMSR, WRMSR

### rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General Protection, #GP	X	X	X	The value in ECX specified an unimplemented performance counter number.
		X	X	CPL was not 0 and CR4.PCE = 0.

## RDTSC

## Read Time-Stamp Counter

Loads the value of the processor’s 64-bit time-stamp counter into registers EDX:EAX.

The time-stamp counter (TSC) is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC.

The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register. This instruction ignores operand size.

When the time-stamp disable flag (TSD) in CR4 is set to 1, the RDTSC instruction can only be used at privilege level 0. If the TSD flag is 0, this instruction can be used at any privilege level.

This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the time-stamp counter is read.

The behavior of the RDTSC instruction is implementation dependent. The TSC counts at a constant rate, but may be affected by power management events (such as frequency changes), depending on the processor implementation. If CPUID Fn8000\_0007\_EDX[TscInvariant] = 1, then the TSC rate is ensured to be invariant across all P-States, C-States, and stop-grant transitions (such as STPCLK Throttling); therefore, the TSC is suitable for use as a source of time. Consult the *BIOS and Kernel Developer’s Guide* applicable to your product for information concerning the effect of power management on the TSC.

Support for the RDTSC instruction is indicated by CPUID Fn0000\_0001\_EDX[TSC] = 1 OR CPUID Fn8000\_0001\_EDX[TSC] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

### Instruction Encoding

Mnemonic	Opcode	Description
RDTSC	0F 31	Copy the time-stamp counter into EDX:EAX.

### Related Instructions

RDTSCP, RDMSR, WRMSR

### rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protecte d	Cause of Exception
Invalid opcode, #UD	X	X	X	The RDTSC instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[TSC] = 0 OR CPUID Fn8000_0001_EDX[TSC] = 0.
General protection, #GP		X	X	CPL was not 0 and CR4.TSD = 1.

## RDTSCP

## Read Time-Stamp Counter and Processor ID

Loads the value of the processor’s 64-bit time-stamp counter into registers EDX:EAX, and loads the value of TSC\_AUX into ECX. This instruction ignores operand size.

The time-stamp counter is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC.

The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register.

The TSC\_AUX value is contained in the low-order 32 bits of the TSC\_AUX register (MSR address C000\_0103h). This MSR is initialized by privileged software to any meaningful value, such as a processor ID, that software wants to associate with the returned TSC value.

When the time-stamp disable flag (TSD) in CR4 is set to 1, the RDTSCP instruction can only be used at privilege level 0. If the TSD flag is 0, this instruction can be used at any privilege level.

Unlike the RDTSCL instruction, RDTSCP forces all older instructions to retire before reading the time-stamp counter.

The behavior of the RDTSCP instruction is implementation dependent. The TSC counts at a constant rate, but may be affected by power management events (such as frequency changes), depending on the processor implementation. If CPUID Fn8000\_0007\_EDX[TscInvariant] = 1, then the TSC rate is ensured to be invariant across all P-States, C-States, and stop-grant transitions (such as STPCLK Throttling); therefore, the TSC is suitable for use as a source of time. Consult the *BIOS and Kernel Developer’s Guide* applicable to your product for information concerning the effect of power management on the TSC.

Support for the RDTSCP instruction is indicated by CPUID Fn8000\_0001\_EDX[RDTSCP] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

### Instruction Encoding

Mnemonic	Opcode	Description
RDTSCP	0F 01 F9	Copy the time-stamp counter into EDX:EAX and the TSC_AUX register into ECX.

### Related Instructions

RDTSC

### rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protecte d	Cause of Exception
Invalid opcode, #UD	X	X	X	The RDTSCP instruction is not supported, as indicated by CPUID Fn8000_0001_EDX[RDTSCP] = 0.
General protection, #GP		X	X	CPL was not 0 and CR4.TSD = 1.

## RSM

## Resume from System Management Mode

Resumes an operating system or application procedure previously interrupted by a system management interrupt (SMI). The processor state is restored from the information saved when the SMI was taken. The processor goes into a shutdown state if it detects invalid state information in the system management mode (SMM) save area during RSM.

RSM will shut down if any of the following conditions are found in the save map (SSM):

- An illegal combination of flags in CR0 (CR0.PG = 1 and CR0.PE = 0, or CR0.NW = 1 and CR0.CD = 0).
- A reserved bit in CR0, CR3, CR4, DR6, DR7, or the extended feature enable register (EFER) is set to 1.
- The following bit combination occurs: EFER.LME = 1, CR0.PG = 1, CR4.PAE = 0.
- The following bit combination occurs: EFER.LME = 1, CR0.PG = 1, CR4.PAE = 1, CS.D = 1, CS.L = 1.
- SMM revision field has been modified.

RSM cannot modify EFER.SVME. Attempts to do so are ignored.

When EFER.SVME is 1, RSM reloads the four PDPEs (through the incoming CR3) when returning to a mode that has legacy PAE mode paging enabled.

When EFER.SVME is 1, the RSM instruction is permitted to return to paged real mode (i.e., CR0.PE=0 and CR0.PG=1).

The AMD64 architecture uses a new 64-bit SMM state-save memory image. This 64-bit save-state map is used in all modes, regardless of mode. See “System-Management Mode” in Volume 2 for details.

Mnemonic	Opcode	Description
RSM	0F AA	Resume operation of an interrupted program.

### Related Instructions

None

## rFLAGS Affected

All flags are restored from the state-save map (SSM).

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The processor was not in System Management Mode (SMM).

**SGDT****Store Global Descriptor Table Register**

Stores the global descriptor table register (GDTR) into the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode, it is 10 bytes. In all modes, operand-size prefixes are ignored.

In non-64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 4 bytes specify the 32-bit base address.

In 64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 8 bytes specify the 64-bit base address.

This instruction is intended for use in operating system software, but it can be used at any privilege level.

Mnemonic	Opcode	Description
SGDT <i>mem16:32</i>	0F 01 /0	Store global descriptor table register to memory.
SGDT <i>mem16:64</i>	0F 01 /0	Store global descriptor table register to memory.

**Related Instructions**

SIDT, SLDT, STR, LGDT, LIDT, LLDT, LTR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**SIDT****Store Interrupt Descriptor Table Register**

Stores the interrupt descriptor table register (IDTR) in the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode it is 10 bytes. In all modes, operand-size prefixes are ignored.

In non-64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 4 bytes specify the 32-bit base address.

In 64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 8 bytes specify the 64-bit base address.

This instruction is intended for use in operating system software, but it can be used at any privilege level.

Mnemonic	Opcode	Description
SIDT <i>mem16:32</i>	0F 01 /1	Store interrupt descriptor table register to memory.
SIDT <i>mem16:64</i>	0F 01 /1	Store interrupt descriptor table register to memory.

**Related Instructions**

SGDT, SLDT, STR, LGDT, LIDT, LLDT, LTR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

**SKINIT****Secure Init and Jump with Attestation**

Securely reinitializes the cpu, allowing for the startup of trusted software (such as a VMM). The code to be executed after reinitialization can be verified based on a secure hash comparison. SKINIT takes the physical base address of the SLB as its only input operand, in EAX. The SLB must be structured as described in “Secure Loader Block” on page 499 of the *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*, order# 24593, and is assumed to contain the code for a Secure Loader (SL).

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001\_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
SKINIT EAX	OF 01 DE	Secure initialization and jump, with attestation.

**Action**

```

IF ((EFER.SVMEN == 0) && !(CPUID 8000_0001.ECX[SKINIT]) || (!PROTECTED_MODE))

    EXCEPTION [#UD]           // This instruction can only be executed
                                // in protected mode with SVM enabled.

IF (CPL != 0)                  // This instruction is only allowed at CPL 0.

    EXCEPTION [#GP]

Initialize processor state as for an INIT signal
CR0.PE = 1

CS.sel = 0x0008
CS.attr = 32-bit code, read/execute
CS.base = 0
CS.limit = 0xFFFFFFFF

SS.sel = 0x0010
SS.attr = 32-bit stack, read/write, expand up
SS.base = 0
SS.limit = 0xFFFFFFFF

EAX = EAX & 0xFFFF0000 // Form SLB base address.
EDX = family/model/stepping
ESP = EAX + 0x00010000 // Initial SL stack.
Clear GPRs other than EAX, EDX, ESP

EFER = 0
VM_CR.DPD = 1

```

VM\_CR.R\_INIT = 1  
 VM\_CR.DIS\_A20M = 1

Enable SL\_DEV, to protect 64Kbyte of physical memory starting at the physical address in EAX

GIF = 0

Read the SL length from offset 0x0002 in the SLB

Copy the SL image to the TPM for attestation

Read the SL entrypoint offset from offset 0x0000 in the SLB

Jump to the SL entrypoint, at EIP = EAX+entrypoint offset

## Related Instructions

None.

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	Secure Virtual Machine was not enabled (EFER.SVME=0) and both of the following conditions were true: <ul style="list-style-type: none"> <li>SVM-Lock is not available, as indicated by CPUID Fn8000_000A_EDX[SVML] = 0.</li> <li>DEV is not available, as indicated by CPUID Fn8000_0001_ECX[SKINIT] = 0.</li> </ul>
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.

**SLDT****Store Local Descriptor Table Register**

Stores the local descriptor table (LDT) selector to a register or memory destination operand.

If the destination is a register, the selector is zero-extended into a 16-, 32-, or 64-bit general purpose register, depending on operand size.

If the destination operand is a memory location, the segment selector is written to memory as a 16-bit value, regardless of operand size.

This SLDT instruction can only be used in protected mode, but it can be executed at any privilege level.

Mnemonic	Opcode	Description
SLDT reg16	0F 00 /0	Store the segment selector from the local descriptor table register to a 16-bit register.
SLDT reg32	0F 00 /0	Store the segment selector from the local descriptor table register to a 32-bit register.
SLDT reg64	0F 00 /0	Store the segment selector from the local descriptor table register to a 64-bit register.
SLDT mem16	0F 00 /0	Store the segment selector from the local descriptor table register to a 16-bit memory location.

**Related Instructions**

SIDT, SGDT, STR, LIDT, LGDT, LLDT, LTR

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**SMSW****Store Machine Status Word**

Stores the lower bits of the machine status word (CR0). The target can be a 16-, 32-, or 64-bit register or a 16-bit memory operand.

This instruction is provided for compatibility with early processors.

This instruction can be used at any privilege level (CPL).

Mnemonic	Opcode	Description
SMSW <i>reg16</i>	0F 01 /4	Store the low 16 bits of CR0 to a 16-bit register.
SMSW <i>reg32</i>	0F 01 /4	Store the low 32 bits of CR0 to a 32-bit register.
SMSW <i>reg64</i>	0F 01 /4	Store the entire 64-bit CR0 to a 64-bit register.
SMSW <i>mem16</i>	0F 01 /4	Store the low 16 bits of CR0 to memory.

**Related Instructions**

LMSW, MOV CR*n*

**rFLAGS Affected**

None

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

## STAC

## Set Alignment Check Flag

Sets the Alignment Check flag in the rFLAGS register to one. Support for the STAC instruction is indicated by CPUID Fn07\_EBX[20]=1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

Mnemonic	Opcode	Description
STAC	0F 01 CB	Sets the AC flag

### Related Instructions

CLAC

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			1													
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
		X		Instruction is not supported in virtual mode
			X	Lock prefix (F0h) preceding opcode.
			X	CPL was not 0

**STI****Set Interrupt Flag**

Sets the interrupt flag (IF) in the rFLAGS register to 1, thereby allowing external interrupts received on the INTR input. Interrupts received on the non-maskable interrupt (NMI) input are not affected by this instruction.

In real mode, this instruction sets IF to 1.

In protected mode and virtual-8086-mode, this instruction is IOPL-sensitive. If the CPL is less than or equal to the rFLAGS.IOPL field, the instruction sets IF to 1.

In protected mode, if  $\text{IOPL} < 3$ ,  $\text{CPL} = 3$ , and protected mode virtual interrupts are enabled ( $\text{CR4.PVI} = 1$ ), then the instruction instead sets rFLAGS.VIF to 1. If none of these conditions apply, the processor raises a general protection exception (#GP). For more information, see “Protected Mode Virtual Interrupts” in Volume 2.

In virtual-8086 mode, if  $\text{IOPL} < 3$  and the virtual-8086-mode extensions are enabled ( $\text{CR4.VME} = 1$ ), the STI instruction instead sets the virtual interrupt flag (rFLAGS.VIF) to 1.

If STI sets the IF flag and IF was initially clear, then interrupts are not enabled until after the instruction following STI. Thus, if IF is 0, this code will not allow an INTR to happen:

```
STI  
CLI
```

In the following sequence, INTR will be allowed to happen only after the NOP.

```
STI  
NOP  
CLI
```

If STI sets the VIF flag and VIP is already set, a #GP fault will be generated.

See “Virtual-8086 Mode Extensions” in Volume 2 for more information about IOPL-sensitive instructions.

Mnemonic	Opcode	Description
STI	FB	Set interrupt flag (IF) to 1.

## Action

```

IF (CPL <= IOPL)
    RFLAGS.IF = 1

ELSIF (((VIRTUAL_MODE) && (CR4.VME == 1))
        || ((PROTECTED_MODE) && (CR4.PVI == 1) && (CPL == 3)))
{
    IF (RFLAGS.VIP == 1)
        EXCEPTION[#GP(0)]
    RFLAGS.VIF = 1
}
ELSE
    EXCEPTION[#GP(0)]

```

## Related Instructions

CLI

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		M								M						
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. M (modified) is either set to one or cleared to zero. Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		The CPL was greater than the IOPL and virtual-mode extensions were not enabled (CR4.VME = 0).
			X	The CPL was greater than the IOPL and either the CPL was not 3 or protected-mode virtual interrupts were not enabled (CR4.PVI = 0).
		X	X	This instruction would set RFLAGS.VIF to 1 and RFLAGS.VIP was already 1.

**STGI****Set Global Interrupt Flag**

Sets the global interrupt flag (GIF) to 1. While GIF is zero, all external interrupts are disabled.

This is a Secure Virtual Machine (SVM) instruction.

Attempted execution of this instruction causes a #UD exception if SVM is not enabled and neither SVM Lock nor the device exclusion vector (DEV) are supported. Support for SVM Lock is indicated by CPUID Fn8000\_000A\_EDX[SVML] = 1. Support for DEV is part of the SKINIT architecture and is indicated by CPUID Fn8000\_0001\_ECX[SKINIT] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

For information on enabling SVM, see “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume-2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
STGI	0F 01 DC	Sets the global interrupt flag (GIF).

**Related Instructions**

CLGI

**rFLAGS Affected**

None.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	Secure Virtual Machine was not enabled (EFER.SVME=0) and both of the following conditions were true: <ul style="list-style-type: none"> <li>• SVM Lock is not available, as indicated by CPUID Fn8000_000A_EDX[SVML] = 0.</li> <li>• DEV is not available, as indicated by CPUID Fn8000_0001_ECX[SKINIT] = 0.</li> </ul>
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.

## STR

## Store Task Register

Stores the task register (TR) selector to a register or memory destination operand.

If the destination is a register, the selector is zero-extended into a 16-, 32-, or 64-bit general purpose register, depending on the operand size.

If the destination is a memory location, the segment selector is written to memory as a 16-bit value, regardless of operand size.

The STR instruction can only be used in protected mode, but it can be used at any privilege level.

Mnemonic	Opcode	Description
STR reg16	0F 00 /1	Store the segment selector from the task register to a 16-bit general-purpose register.
STR reg32	0F 00 /1	Store the segment selector from the task register to a 32-bit general-purpose register.
STR reg64	0F 00 /1	Store the segment selector from the task register to a 64-bit general-purpose register.
STR mem16	0F 00 /1	Store the segment selector from the task register to a 16-bit memory location.

### Related Instructions

LGDT, LIDT, LLDT, LTR, SIDT, SGDT, SLDT

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## SWAPGS

## Swap GS Register with KernelGSbase MSR

Provides a fast method for system software to load a pointer to system data structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction, an interrupt or an exception. Prior to returning to application software, SWAPGS can be used to restore the application data pointer that was replaced by the system data-structure pointer.

This instruction can only be executed in 64-bit mode. Executing SWAPGS in any other mode generates an undefined opcode exception.

The SWAPGS instruction only exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000\_0102h) with the base-address value located in the hidden portion of the GS selector register (GS.base). This allows the system-kernel software to access kernel data structures by using the GS segment-override prefix during memory references.

The address stored in the KernelGSbase MSR must be in canonical form. The WRMSR instruction used to load the KernelGSbase MSR causes a general-protection exception if the address loaded is not in canonical form. The SWAPGS instruction itself does not perform a canonical check.

This instruction is only valid in 64-bit mode at CPL 0. A general protection exception (#GP) is generated if this instruction is executed at any other privilege level.

For additional information about this instruction, refer to “System-Management Instructions” in Volume 2.

### Examples

At a kernel entry point, the OS uses SwapGS to obtain a pointer to kernel data structures and simultaneously save the user's GS base. Upon exit, it uses SwapGS to restore the user's GS base:

```
SystemCallEntryPoint:
SwapGS          ; get kernel pointer, save user GSbase
mov gs:[SavedUserRSP], rsp    ; save user's stack pointer
mov rsp, gs:[KernelStackPtr]  ; set up kernel stack
push rax           ; now save user GPRs on kernel stack
                     ; perform system service
                     .
                     .
SwapGS          ; restore user GS, save kernel pointer
```

Mnemonic	Opcode	Description
SWAPGS	0F 01 F8	Exchange GS base with KernelGSBase MSR. (Invalid in legacy and compatibility modes.)

### Related Instructions

None

## rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	This instruction was executed in legacy or compatibility mode.
General protection, #GP			X	CPL was not 0.

## SYSCALL

## Fast System Call

Transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model.

The SYSCALL and SYSRET instructions are low-latency system call and return control-transfer instructions, which assume that the operating system implements a flat-segment memory model. By eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions), calls to and returns from the operating system are greatly simplified. These instructions can be used in protected mode and are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-segment memory model.

This instruction has been optimized by reducing the number of checks and memory references that are normally made so that a call or return takes considerably fewer clock cycles than the CALL FAR /RET FAR instruction method.

It is assumed that the base, limit, and attributes of the Code Segment will remain flat for all processes and for the operating system, and that only the current privilege level for the selector of the calling process should be changed from a current privilege level of 3 to a new privilege level of 0. It is also assumed (but not checked) that the RPL of the SYSCALL and SYSRET target selectors are set to 0 and 3, respectively.

SYSCALL sets the CPL to 0, regardless of the values of bits 33:32 of the STAR register. There are no permission checks based on the CPL, real mode, or virtual-8086 mode. SYSCALL and SYSRET must be enabled by setting EFER.SCE to 1.

It is the responsibility of the operating system to keep the descriptors in memory that correspond to the CS and SS selectors loaded by the SYSCALL and SYSRET instructions consistent with the segment base, limit, and attribute values forced by these instructions.

**Legacy x86 Mode.** In legacy x86 mode, when SYSCALL is executed, the EIP of the instruction following the SYSCALL is copied into the ECX register. Bits 31:0 of the SYSCALL/SYSRET target address register (STAR) are copied into the EIP register. (The STAR register is model-specific register C000\_0081h.)

New selectors are loaded, without permission checking (see above), as follows:

- Bits 47:32 of the STAR register specify the selector that is copied into the CS register.
- Bits 47:32 of the STAR register + 8 specify the selector that is copied into the SS register.
- The CS\_base and the SS\_base are both forced to zero.
- The CS\_limit and the SS\_limit are both forced to 4 Gbyte.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

**Long Mode.** When long mode is activated, the behavior of the SYSCALL instruction depends on whether the calling software is in 64-bit mode or compatibility mode. In 64-bit mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from LSTAR bits 63:0. (The LSTAR register is model-specific register C000\_0082h.) In compatibility mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from CSTAR bits 63:0. (The CSTAR register is model-specific register C000\_0083h.)

New selectors are loaded, without permission checking (see above), as follows:

- Bits 47:32 of the STAR register specify the selector that is copied into the CS register.
- Bits 47:32 of the STAR register + 8 specify the selector that is copied into the SS register.
- The CS\_base and the SS\_base are both forced to zero.
- The CS\_limit and the SS\_limit are both forced to 4 Gbyte.
- The CS segment attributes are set to execute/read 64-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 64-bit stack referenced by RSP.

The WRMSR instruction loads the target RIP into the LSTAR and CSTAR registers. If an RIP written by WRMSR is not in canonical form, a general-protection exception (#GP) occurs.

How SYSCALL and SYSRET handle rFLAGS, depends on the processor's operating mode.

In legacy mode, SYSCALL treats EFLAGS as follows:

- EFLAGS.IF is cleared to 0.
- EFLAGS.RF is cleared to 0.
- EFLAGS.VM is cleared to 0.

In long mode, SYSCALL treats RFLAGS as follows:

- The current value of RFLAGS is saved in R11.
- RFLAGS is masked using the value stored in SYSCALL\_FLAG\_MASK.
- RFLAGS.RF is cleared to 0.

For further details on the SYSCALL and SYSRET instructions and their associated MSR registers (STAR, LSTAR, CSTAR, and SYSCALL\_FLAG\_MASK), see “Fast System Call and Return” in Volume 2.

Support for the SYSCALL instruction is indicated by CPUID Fn8000\_0001\_EDX[SysCallSysRet] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

## Instruction Encoding

Mnemonic	Opcode	Description
SYSCALL	0F 05	Call operating system.

### Action

// See "Pseudocode Definition" on page 57.

SYSCALL\_START:

```
IF (MSR_EFER.SCE == 0)           // Check if syscall/sysret are enabled.
    EXCEPTION [#UD]

IF (LONG_MODE)
    SYSCALL_LONG_MODE
ELSE // (LEGACY_MODE)
    SYSCALL_LEGACY_MODE
```

SYSCALL\_LONG\_MODE:

```
RCX.q = next_RIP
R11.q = RFLAGS      // with rf cleared

IF (64BIT_MODE)
    temp_RIP.q = MSR_LSTAR
ELSE // (COMPATIBILITY_MODE)
    temp_RIP.q = MSR_CSTAR

CS.sel    = MSR_STAR.SYSCALL_CS AND 0xFFFFC
CS.attr   = 64-bit code,dpl0 // Always switch to 64-bit mode in long mode.
CS.base   = 0x00000000
CS.limit  = 0xFFFFFFFF
```

```
SS.sel    = MSR_STAR.SYSCALL_CS + 8
SS.attr   = 64-bit stack,dpl0
SS.base   = 0x00000000
SS.limit  = 0xFFFFFFFF
```

```
RFLAGS = RFLAGS AND ~MSR_SFMASK
RFLAGS.RF = 0
```

```
CPL = 0
```

```
RIP = temp_RIP
EXIT
```

SYSCALL\_LEGACY\_MODE:

```

RCX.d = next_RIP

temp_RIP.d = MSR_STAR.EIP

CS.sel = MSR_STAR.SYSCALL_CS AND 0xFFFFC
CS.attr = 32-bit code,dpl0 // Always switch to 32-bit mode in legacy mode.
CS.base = 0x00000000
CS.limit = 0xFFFFFFFF

SS.sel = MSR_STAR.SYSCALL_CS + 8
SS.attr = 32-bit stack,dpl0
SS.base = 0x00000000
SS.limit = 0xFFFFFFFF

RFLAGS.VM,IF,RF=0

CPL = 0

RIP = temp_RIP
EXIT

```

## Related Instructions

SYSRET, SYSENTER, SYSEXIT

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M	0	0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSCALL and SYSRET instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[SysCallSysRet] = 0.
	X	X	X	The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.)

## SYSENTER

## System Call

Transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model. This instruction is valid only in legacy mode.

Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction, as well as the CS and SS selectors of the called and returned procedures:

- MSR\_SYSENTER\_CS: Contains the CS selector of the called procedure. The SS selector is set to MSR\_SYSENTER\_CS + 8.
- MSR\_SYSENTER\_ESP: Contains the called procedure's stack pointer.
- MSR\_SYSENTER\_EIP: Contains the offset into the CS of the called procedure.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 CALL instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are forced to 0.
- The CS and SS limit values are forced to 4 Gbytes.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above.

The return EIP and application stack are not saved by this instruction. System software must explicitly save that information.

An invalid-opcode exception occurs if this instruction is used in long mode. Software should use the SYSCALL (and SYSRET) instructions in long mode. If SYSENTER is used in real mode, a #GP is raised.

For additional information on this instruction, see “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2.

Support for the SYSENTER instruction is indicated by CPUID Fn0000\_0001\_EDX[SysEnterSysExit] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

### Instruction Encoding

Mnemonic	Opcode	Description
SYSENTER	0F 34	Call operating system.

## Related Instructions

SYSCALL, SYSEXIT, SYSRET

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
				0						0						
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSENTER and SYSEXIT instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SysEnterSysExit] = 0.
			X	This instruction is not recognized in long mode.
General protection, #GP	X			This instruction is not recognized in real mode.
		X	X	MSR_SYSENTER_CS was a null selector.

## SYSEXIT

## System Return

Returns from the operating system to an application. It is a low-latency system return instruction designed for use by system and application software implementing a flat-segment memory model.

This is a privileged instruction. The current privilege level must be zero to execute this instruction. An invalid-opcode exception occurs if this instruction is used in long mode. Software should use the SYSRET (and SYSCALL) instructions when running in long mode.

When a system procedure performs a SYSEXIT back to application software, the CS selector is updated to point to the second descriptor entry after the SYSENTER CS value (MSR SYSENTER\_CS+16). The SS selector is updated to point to the third descriptor entry after the SYSENTER CS value (MSR SYSENTER\_CS+24). The CPL is forced to 3, as are the descriptor privilege levels.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are forced to 0.
- The CS and SS limit values are forced to 4 Gbytes.
- The CS segment attributes are set to 32-bit read/execute at CPL 3.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above.

The following additional actions result from executing SYSEXIT:

- EIP is loaded from EDX.
- ESP is loaded from ECX.

System software must explicitly load the return address and application software-stack pointer into the EDX and ECX registers prior to executing SYSEXIT.

For additional information on this instruction, see “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2.

Support for the SYSEXIT instruction is indicated by CPUID Fn0000\_0001\_EDX[SysEnterSysExit] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

## Instruction Encoding

Mnemonic	Opcode	Description
SYSEXIT	0F 35	Return from operating system to application.

## Related Instructions

SYSCALL, SYSENTER, SYSRET

## rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
					0											
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSENTER and SYSEXIT instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SysEnterSysExit] = 0.
			X	This instruction is not recognized in long mode.
General protection, #GP	X	X		This instruction is only recognized in protected mode.
			X	CPL was not 0.
			X	MSR_SYSENTER_CS was a null selector.

## SYSRET

## Fast System Return

Returns from the operating system to an application. It is a low-latency system return instruction designed for use by system and application software implementing a flat segmentation memory model.

The SYSCALL and SYSRET instructions are low-latency system call and return control-transfer instructions that assume that the operating system implements a flat-segment memory model. By eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions), calls to and returns from the operating system are greatly simplified. These instructions can be used in protected mode and are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-segment memory model.

This instruction has been optimized by reducing the number of checks and memory references that are normally made so that a call or return takes substantially fewer internal clock cycles when compared to the CALL/RET instruction method.

It is assumed that the base, limit, and attributes of the Code Segment will remain flat for all processes and for the operating system, and that only the current privilege level for the selector of the calling process should be changed from a current privilege level of 0 to a new privilege level of 3. It is also assumed (but not checked) that the RPL of the SYSCALL and SYSRET target selectors are set to 0 and 3, respectively.

SYSRET sets the CPL to 3, regardless of the values of bits 49:48 of the star register. SYSRET can only be executed in protected mode at CPL 0. SYSCALL and SYSRET must be enabled by setting EFER.SCE to 1.

It is the responsibility of the operating system to keep the descriptors in memory that correspond to the CS and SS selectors loaded by the SYSCALL and SYSRET instructions consistent with the segment base, limit, and attribute values forced by these instructions.

When a system procedure performs a SYSRET back to application software, the CS selector is updated from bits 63:50 of the STAR register (STAR.SYSRET\_CS) as follows:

- If the return is to 32-bit mode (legacy or compatibility), CS is updated with the value of STAR.SYSRET\_CS.
- If the return is to 64-bit mode, CS is updated with the value of STAR.SYSRET\_CS + 16.

In both cases, the CPL is forced to 3, effectively ignoring STAR bits 49:48. The SS selector is updated to point to the next descriptor-table entry after the CS descriptor (STAR.SYSRET\_CS + 8), and its RPL is not forced to 3.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS base value is forced to 0.
- The CS limit value is forced to 4 Gbytes.

- The CS segment attributes are set to execute-read 32 bits or 64 bits (see below).
- The SS segment base, limit, and attributes are not modified.

When SYSCALLed system software is running in 64-bit mode, it has been entered from either 64-bit mode or compatibility mode. The corresponding SYSRET needs to know the mode to which it must return. Executing SYSRET in non-64-bit mode or with a 16- or 32-bit operand size returns to 32-bit mode with a 32-bit stack pointer. Executing SYSRET in 64-bit mode with a 64-bit operand size returns to 64-bit mode with a 64-bit stack pointer.

The instruction pointer is updated with the return address based on the operating mode in which SYSRET is executed:

- If returning to 64-bit mode, SYSRET loads RIP with the value of RCX.
- If returning to 32-bit mode, SYSRET loads EIP with the value of ECX.

How SYSRET handles RFLAGS depends on the processor's operating mode:

- If executed in 64-bit mode, SYSRET loads the lower-32 RFLAGS bits from R11[31:0] and clears the upper 32 RFLAGS bits.
- If executed in legacy mode or compatibility mode, SYSRET sets EFLAGS.IF.

For further details on the SYSCALL and SYSRET instructions and their associated MSR registers (STAR, LSTAR, and CSTAR), see “Fast System Call and Return” in Volume 2.

Support for the SYSRET instruction is indicated by CPUID Fn8000\_0001\_EDX[SysCallSysRet] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

## Instruction Encoding

Mnemonic	Opcode	Description
SYSRET	0F 07	Return from operating system.

### Action

// See “Pseudocode Definition” on page 57.

SYSRET\_START:

```

IF (MSR_EFER.SCE == 0)           // Check if syscall/sysret are enabled.
    EXCEPTION [#UD]

IF (( !PROTECTED_MODE) || (CPL != 0))
    EXCEPTION [#GP(0)]          // SYSRET requires protected mode, cpl0

IF (64BIT_MODE)
    SYSRET_64BIT_MODE
ELSE // (!64BIT_MODE)
```

```
SYSRET_NON_64BIT_MODE

SYSRET_64BIT_MODE:

    IF (OPERAND_SIZE == 64)           // Return to 64-bit mode.
    {
        CS.sel     = MSR_STAR.SYSRET_CS + 16) OR 3
        CS.base    = 0x00000000
        CS.limit   = 0xFFFFFFFF
        CS.attr    = 64-bit code,dpl3

        temp_RIP.q = RCX
    }
    ELSE                         // Return to 32-bit compatibility mode.
    {
        CS.sel     = MSR_STAR.SYSRET_CS OR 3
        CS.base    = 0x00000000
        CS.limit   = 0xFFFFFFFF
        CS.attr    = 32-bit code,dpl3

        temp_RIP.d = RCX
    }

    SS.sel = MSR_STAR.SYSRET_CS + 8      // SS selector is changed,
                                         // SS base, limit, attributes unchanged.

    RFLAGS.q = R11          // RF=0,VM=0
    CPL = 3

    RIP = temp_RIP
    EXIT

SYSRET_NON_64BIT_MODE:

    CS.sel     = MSR_STAR.SYSRET_CS OR 3 // Return to 32-bit legacy protected mode.
    CS.base    = 0x00000000
    CS.limit   = 0xFFFFFFFF
    CS.attr    = 32-bit code,dpl3

    temp_RIP.d = RCX

    SS.sel = MSR_STAR.SYSRET_CS + 8      // SS selector is changed.
                                         // SS base, limit, attributes unchanged.

    RFLAGS.IF = 1
    CPL = 3

    RIP = temp_RIP
    EXIT
```

## Related Instructions

SYSCALL, SYSENTER, SYSEXIT

### rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M		0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSCALL and SYSRET instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[SysCallSysRet] = 0.
	X	X	X	The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.)
General protection, #GP	X	X		This instruction is only recognized in protected mode.
			X	CPL was not 0.

**VERR****Verify Segment for Reads**

Verifies whether a code or data segment specified by the segment selector in the 16-bit register or memory operand is readable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is readable. Otherwise, ZF is cleared.

A segment is readable if all of the following apply:

- the selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the segment is a data segment or readable code segment.
- the descriptor DPL is greater than or equal to both the CPL and RPL, or the segment is a conforming code segment.

The processor does not recognize the VERR instruction in real or virtual-8086 mode.

Mnemonic	Opcode	Description
VERR reg/mem16	0F 00 /4	Set the zero flag (ZF) to 1 if the segment selected can be read.

**Related Instructions**

ARPL, LAR, LSL, VERW

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
														M		
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or is non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

**VERW****Verify Segment for Write**

Verifies whether a data segment specified by the segment selector in the 16-bit register or memory operand is writable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is writable. Otherwise, ZF is cleared.

A segment is writable if all of the following apply:

- the selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the segment is a writable data segment.
- the descriptor DPL is greater than or equal to both the CPL and RPL.

The processor does not recognize the VERW instruction in real or virtual-8086 mode.

Mnemonic	Opcode	Description
VERW reg/mem16	OF 00 /5	Set the zero flag (ZF) to 1 if the segment selected can be written.

**Related Instructions**

ARPL, LAR, LSL, VERR

**rFLAGS Affected**

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
														M		
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

**Note:** Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to access memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

## VMLOAD

## Load State from VMCB

Loads a subset of processor state from the VMCB specified by the system-physical address in the rAX register. The portion of RAX used to form the address is determined by the effective address size.

The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001\_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
VMLOAD rAX	0F 01 DA	Load additional state from VMCB.

### Action

```
IF ((MSR_EFER.SVME == 0) || (!PROTECTED_MODE))
    EXCEPTION [#UD]           // This instruction can only be executed in protected
                                // mode with SVM enabled

IF (CPL != 0)                  // This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported system-physical address)
    EXCEPTION [#GP]

Load from a VMCB at system-physical address rAX:
    FS, GS, TR, LDTR (including all hidden state)
    KernelGsBase
    STAR, LSTAR, CSTAR, SFMASK
    SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP
```

### Related Instructions

VMSAVE

### rFLAGS Affected

None.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001(ECX[SVM] = 0).
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		The instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.
			X	rAX referenced a physical address above the maximum supported physical address.
			X	The address in rAX was not aligned on a 4Kbyte boundary.

**VMMCALL****Call VMM**

Provides a mechanism for a guest to explicitly communicate with the VMM by generating a #VMEXIT.

A non-intercepted VMMCALL unconditionally raises a #UD exception.

VMMCALL is not restricted to either protected mode or CPL zero.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001(ECX[SVM]) = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
VMMCALL	0F 01 D9	Explicit communication with the VMM.

**Related Instructions**

None.

**rFLAGS Affected**

None.

**Exceptions**

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001(ECX[SVM]) = 0.
	X	X	X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X	X	VMMCALL was not intercepted.

## VMRUN

## Run Virtual Machine

Starts execution of a guest instruction stream. The physical address of the *virtual machine control block* (VMCB) describing the guest is taken from the rAX register (the portion of RAX used to form the address is determined by the effective address size). The physical address of the VMCB must be aligned on a 4K-byte boundary.

VMRUN saves a subset of host processor state to the host state-save area specified by the physical address in the VM\_HSAVE\_PA MSR. VMRUN then loads guest processor state (and control information) from the VMCB at the physical address specified in rAX. The processor then executes guest instructions until one of several *intercept* events (specified in the VMCB) is triggered. When an intercept event occurs, the processor stores a snapshot of the guest state back into the VMCB, reloads the host state, and continues execution of host code at the instruction following the VMRUN instruction.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001\_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

The VMRUN instruction is not supported in System Management Mode. Processor behavior resulting from an attempt to execute this instruction from within the SMM handler is undefined.

### Instruction Encoding

Mnemonic	Opcode	Description
VMRUN rAX	0F 01 D8	Performs a world-switch to guest.

### Action

```

IF ((MSR_EFER.SVME == 0) || (!PROTECTED_MODE))
    EXCEPTION [#UD]           // This instruction can only be executed in protected
                                // mode with SVM enabled

IF (CPL != 0)                  // This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported physical address)
    EXCEPTION [#GP]

IF (intercepted(VMRUN))
    #VMEXIT (VMRUN)
remember VMCB address (delivered in rAX) for next #VMEXIT
save host state to physical memory indicated in the VM_HSAVE_PA MSR:
    ES.sel
    CS.sel
    SS.sel

```

```
DS.sel
GDTR.{base,limit}
IDTR.{base,limit}
EFER
CR0
CR4
CR3
// host CR2 is not saved
RFLAGS
RIP
RSP
RAX

from the VMCB at physical address rAX, load control information:
    intercept vector
    TSC_OFFSET
    interrupt control (v_irq, v_intr_*, v_tpr)
    EVENTINJ field
    ASID

IF(nested paging supported)
    NP_ENABLE
    IF (NP_ENABLE == 1)
        nCR3

from the VMCB at physical address rAX, load guest state:
    ES.{base,limit,attr,sel}
    CS.{base,limit,attr,sel}
    SS.{base,limit,attr,sel}
    DS.{base,limit,attr,sel}
    GDTR.{base,limit}
    IDTR.{base,limit}
    EFER
    CR0
    CR4
    CR3
    CR2
    IF (NP_ENABLE == 1)
        gPAT           // Leaves host hPAT register unchanged.
        RFLAGS
        RIP
        RSP
        RAX
        DR7
        DR6
        CPL           // 0 for real mode, 3 for v86 mode, else as loaded.
        INTERRUPT_SHADOW

IF (LBR virtualization supported)
    LBR_VIRTUALIZATION_ENABLE
    IF (LBR_VIRTUALIZATION_ENABLE == 1)
```

```
save LBR state to the host save area
    DBGCTL
    BR_FROM
    BR_TO
    LASTEXCP_FROM
    LASTEXCP_TO
load LBR state from the VMCB
    DBGCTL
    BR_FROM
    BR_TO
    LASTEXCP_FROM
    LASTEXCP_TO

IF (guest state consistency checks fail)
#VMEXIT(INVALID)

Execute command stored in TLB_CONTROL.

GIF = 1          // allow interrupts in the guest
IF (EVENTINJ.V)
    cause exception/interrupt in guest
else
    jump to first guest instruction
```

Upon #VMEXIT, the processor performs the following actions in order to return to the host execution context:

```
GIF = 0
save guest state to VMCB:
    ES.{base,limit,attr,sel}
    CS.{base,limit,attr,sel}
    SS.{base,limit,attr,sel}
    DS.{base,limit,attr,sel}
    GDTR.{base,limit}
    IDTR.{base,limit}
    EFER
    CR4
    CR3
    CR2
    CR0
    if (nested paging enabled)
        gPAT
    RFLAGS
    RIP
    RSP
    RAX
    DR7
    DR6
    CPL
    INTERRUPT_SHADOW
save additional state and intercept information:
    V_IRQ, V_TPR
```

```
EXITCODE
EXITINFO1
EXITINFO2
EXITINTINFO
clear EVENTINJ field in VMCB

prepare for host mode by clearing internal processor state bits:
    clear intercepts
    clear v_irq
    clear v_intr_masking
    clear tsc_offset
    disable nested paging
    clear ASID to zero

reload host state
    GDTR.{base,limit}
    IDTR.{base,limit}
    EFER
    CR0
    CR0.PE = 1 // saved copy of CR0.PE is ignored
    CR4
    CR3
    if (host is in PAE paging mode)
        reloaded host PDPEs
    // Do not reload host CR2 or PAT
    RFLAGS
    RIP
    RSP
    RAX
    DR7 = "all disabled"
    CPL = 0
    ES.sel; reload segment descriptor from GDT
    CS.sel; reload segment descriptor from GDT
    SS.sel; reload segment descriptor from GDT
    DS.sel; reload segment descriptor from GDT

if (LBR virtualization supported)
    LBR_VIRTUALIZATION_ENABLE
    if (LBR_VIRTUALIZATION_ENABLE == 1)
        save LBR state to the VMCB:
            DBGCTL
            BR_FROM
            BR_TO
            LASTEXCP_FROM
            LASTEXCP_TO
        load LBR state from the host save area:
            DBGCTL
            BR_FROM
            BR_TO
            LASTEXCP_FROM
            LASTEXCP_TO
```

```

if (illegal host state loaded, or exception while loading host state)
    shutdown
else
    execute first host instruction following the VMRUN

```

## Related Instructions

VMLOAD, VMSAVE.

## rFLAGS Affected

None.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001(ECX[SVM] = 0).
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		The instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.
			X	rAX referenced a physical address above the maximum supported physical address.
			X	The address in rAX was not aligned on a 4Kbyte boundary.

## VMSAVE

## Save State to VMCB

Stores a subset of the processor state into the VMCB specified by the system-physical address in the rAX register (the portion of RAX used to form the address is determined by the effective address size).

The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000\_0001\_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 158.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

### Instruction Encoding

Mnemonic	Opcode	Description
VMSAVE rAX	OF 01 DB	Save additional guest state to VMCB.

### Action

```

IF ((MSR_EFER.SVME == 0) || (!PROTECTED_MODE))
    EXCEPTION [#UD]           // This instruction can only be executed in protected
                                // mode with SVM enabled

IF (CPL != 0)                  // This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported system-physical address)
    EXCEPTION [#GP]

Store to a VMCB at system-physical address rAX:
    FS, GS, TR, LDTR (including all hidden state)
    KernelGsBase
    STAR, LSTAR, CSTAR, SFMASK
    SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

```

### Related Instructions

VMLOAD

### rFLAGS Affected

None.

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001(ECX[SVM] = 0).
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		The instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not zero.
			X	rAX referenced a physical address above the maximum supported physical address.
			X	The address in rAX was not aligned on a 4Kbyte boundary.

## WBINVD

## Writeback and Invalidate Caches

Writes all modified lines in all levels of cache associated with this processor to main memory and invalidates the caches. This may or may not include lower level caches associated with another processor that shares any level of this processor's cache hierarchy.

CPUID Fn8000\_001D\_EDX[WBINVD]\_xN indicates the behavior of the processor at various levels of the cache hierarchy. If the feature bit is 0, the instruction causes the invalidation of all lower level caches of other processors sharing the designated level of cache. If the feature bit is 1, the instruction does not necessarily cause the invalidation of all lower level caches of other processors sharing the designated level of cache. See Appendix E, “Obtaining Processor Information Via the CPUID Instruction,” on page 601 for more information on using the CPUID function.

The INVD instruction can be used when cache coherence with memory is not important.

This instruction does not invalidate TLB caches.

This is a privileged instruction. The current privilege level of a procedure invalidating the processor's internal caches must be zero.

WBINVD is a serializing instruction.

Mnemonic	Opcode	Description
WBINVD	0F 09	Write modified cache lines to main memory, invalidate internal caches, and trigger external cache flushes.

### Related Instructions

CLFLUSH, INVD

### rFLAGS Affected

None

### Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

## WRMSR

## Write to Model-Specific Register

Writes data to 64-bit model-specific registers (MSRs). These registers are widely used in performance-monitoring and debugging applications, as well as testability and program execution tracing.

This instruction writes the contents of the EDX:EAX register pair into a 64-bit model-specific register specified in the ECX register. The 32 bits in the EDX register are mapped into the high-order bits of the model-specific register and the 32 bits in EAX form the low-order 32 bits.

This instruction must be executed at a privilege level of 0 or a general protection fault #GP(0) will be raised. This exception is also generated if an attempt is made to specify a reserved or unimplemented model-specific register in ECX.

WRMSR is a serializing instruction.

Support for the WRMSR instruction is indicated by CPUID Fn0000\_0001\_EDX[MSR] = 1 OR CPUID Fn8000\_0001\_EDX[MSR] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 158.

The CPUID instruction can provide model information useful in determining the existence of a particular MSR.

See “Model-Specific Registers (MSRs)” in *Volume 2: System Programming*, for more information about model-specific registers, machine check architecture, performance monitoring and debug registers.

Mnemonic	Opcode	Description
WRMSR	0F 30	Write EDX:EAX to the MSR specified by ECX.

### Related Instructions

RDMSR

### rFLAGS Affected

None

## Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The WRMSR instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[MSR] = 0 OR CPUID Fn8000_0001_EDX[MSR] = 0.
General protection, #GP		X	X	CPL was not 0.
	X		X	The value in ECX specifies a reserved or unimplemented MSR address.
	X		X	Writing 1 to any bit that must be zero (MBZ) in the MSR.
	X		X	Writing a non-canonical value to a MSR that can only be written with canonical values.

# Appendix A Opcode and Operand Encodings

This appendix specifies the opcode and operand encodings for each instruction in the AMD64 instruction set. As discussed in Chapter 1, “Instruction Encoding,” the basic operation and implied operand type(s) of an instruction are encoded by the binary value of the opcode byte. The correspondence between an opcode binary value and its meaning is provided by the *opcode map*.

Each opcode map has 256 entries and can encode up to 256 different operations. Since the AMD64 instruction set comprises more than 256 instructions, multiple opcode maps are utilized to encode the instruction set. A particular opcode map is selected using the instruction encoding syntax diagrammed in Figure 1-1 on page 2. For each opcode map, values may be reserved or utilized for purposes other than encoding an instruction operation.

To preserve compatibility with future instruction architectural extensions, reserved opcodes should not be used. If a means to reliably cause an invalid-opcode exception (#UD) is required, software should use one of the UD<sub>x</sub> opcodes. These opcodes are set aside for this purpose and will not be used for future instructions. The UD opcodes are located on the secondary opcode map at code points B9h, 0Bh, and FFh.

The following section provides a key to the notation used in the opcode maps to specify the implied operand types.

## Opcode-Syntax Notation

In the opcode maps which follow, each table entry represents a specific form of an instruction, identifying the instruction by its mnemonic and listing the operand or operands peculiar to that opcode. If a register-based operand is specified by the opcode itself, the operand is represented directly using the register mnemonic as defined in “Summary of Registers and Data Types” on page 38. If the operand is encoded in one or more bytes following the opcode byte, the following special notation is used to represent the operand and its encoding in more generic terms.

This special notation, used exclusively in the opcode maps, is composed of three parts:

- an initial capital letter that represents the operand source / destination (register-based, memory-based, or immediate) and how it is encoded in the instruction (either as an immediate, or via the ModRM.reg, ModRM.{mod,r/m}, or VEX/XOP.vvvv fields). For register-based operands, the initial letter also specifies the register type (General-purpose, MMX, YMM/XMM, debug, or control register).
- one, two, or three letter modifier (in lowercase) that represents the data type (for example, byte, word, quadword, packed single-precision floating-point vector).
- *x*, which indicates for an SSE instruction that the instruction supports both vector sizes (128 bits and 256 bits). The specific vector size is encoded in the VEX/XOP.L field. L=0 indicates 128 bits and L=1 indicates 256 bits.

The following list describes the meaning of each letter that is used in the first position of the operand notation:

- A A far pointer encoded in the instruction. No ModRM byte in the instruction encoding.
- B General-purpose register specified by the VEX or XOP vvvv field.
- C Control register specified by the ModRM.reg field.
- D Debug register specified by the ModRM.reg field.
- E General purpose register or memory operand specified by the r/m field of the ModRM byte. For memory operands, the ModRM byte may be followed by a SIB byte to specify one of the indexed register-indirect addressing forms.
- F rFLAGS register.
- G General purpose register specified by the ModRM.reg field.
- H YMM or XMM register specified by the VEX/XOP.vvvv field.
- I Immediate value encoded in the instruction immediate field.
- J The instruction encoding includes a relative offset that is added to the rIP.
- L YMM or XMM register specified using the most-significant 4 bits of an 8-bit immediate value. In legacy or compatibility mode the most significant bit is ignored.
- M A memory operand specified by the {mod, r/m} field of the ModRM byte. ModRM.mod ≠ 11b.
- M\* A sparse array of memory operands addressed using the VSIB addressing mode. See “VSIB Addressing” in Volume 4.
- N 64-bit MMX register specified by the ModRM.r/m field. The ModRM.mod field must be 11b.
- O The offset of an operand is encoded in the instruction. There is no ModRM byte in the instruction encoding. Indexed register-indirect addressing using the SIB byte is not supported.
- P 64-bit MMX register specified by the ModRM.reg field.
- Q 64-bit MMX-register or memory operand specified by the {mod, r/m} field of the ModRM byte. For memory operands, the ModRM byte may be followed by a SIB byte to specify one of the indexed register-indirect addressing forms.
- R General purpose register specified by the ModRM.r/m field. The ModRM.mod field must be 11b.
- S Segment register specified by the ModRM.reg field.
- U YMM/XMM register specified by the ModRM.r/m field. The ModRM.mod field must be 11b.
- V YMM/XMM register specified by the ModRM.reg field.
- W YMM/XMM register or memory operand specified by the {mod, r/m} field of the ModRM byte. For memory operands, the ModRM byte may be followed by a SIB byte to specify one of the indexed register-indirect addressing forms.

*X* A memory operand addressed by the DS.rSI registers. Used in string instructions.

*Y* A memory operand addressed by the ES.rDI registers. Used in string instructions.

The following list provides the key for the second part of the operand notation:

*a* Two 16-bit or 32-bit memory operands, depending on the effective operand size. Used in the BOUND instruction.

*b* A byte, irrespective of the effective operand size.

*c* A byte or a word, depending on the effective operand size.

*d* A doubleword (32 bits), irrespective of the effective operand size.

*do* A double octword (256 bits), irrespective of the effective operand size.

*i* A 16-bit integer.

*j* A 32-bit integer.

*m* A bit mask of size equal to the source operand.

*mn* Where  $n = 2, 4, 8$ , or 16. A bit mask of size  $n$ .

*o* An octword (128 bits), irrespective of the effective operand size.

*o.q* Operand is either the upper or lower half of a 128-bit value.

*p* A 32-, 48-, 80-bit far pointer, depending on the effective operand size.

*pb* Vector with byte-wide (8-bit) elements (packed byte).

*pd* A double-precision (64-bit) floating-point vector operand (packed double-precision).

*pdw* Vector composed of 32-bit doublewords.

*ph* A half-precision (16-bit) floating-point vector operand (packed half-precision)

*pi* Vector composed of 16-bit integers (packed integer).

*pj* Vector composed of 32-bit integers (packed double integer).

*pk* Vector composed of 8-bit integers (packed half-word integer).

*pq* Vector composed of 64-bit integers (packed quadword integer).

*pqw* Vector composed of 64-bit quadwords (packed quadword).

*ps* A single-precision floating-point vector operand (packed single-precision).

*pw* Vector composed of 16-bit words (packed word).

*q* A quadword (64 bits), irrespective of the effective operand size.

*s* A 6-byte or 10-byte pseudo-descriptor.

*sd* A scalar double-precision floating-point operand (scalar double).

*sj* A scalar doubleword (32-bit) integer operand (scalar double integer).

- ss* A scalar single-precision floating-point operand (scalar single).
- v* A word, doubleword, or quadword (in 64-bit mode), depending on the effective operand size.
- w* A word, irrespective of the effective operand size.
- x* Instruction supports both vector sizes (128 bits or 256 bits). Size is encoded using the VEX/XOP.L field. (L=0: 128 bits; L=1: 256 bits). This symbol may be appended to *ps* or *pd* to represent a packed single- or double-precision floating-point vector of either size; or to *pk*, *pi*, *pj*, or *pq*, to represent a packed 8-bit, 16-bit, 32-bit, or 64-bit packed integer vector of either size.
- y* A doubleword or quadword depending on effective operand size.
- z* A word if the effective operand size is 16 bits, or a doubleword if the effective operand size is 32 or 64 bits.

For some instructions, fields in the ModRM or SIB byte are used as encoding extensions. This is indicated using the following notation:

- /n* A ModRM-byte *reg* field or SIB-byte *base* field, where *n* is a value between zero (000b) and 7 (111b).

For SSE instructions that take scalar operands, VEX/XOP.L field is ignored.

For immediates and memory-based operands, only the size and not the datatype is indicated. Operand widths and datatypes are specified based on the source operands. For instructions where the result overwrites one of the source registers, the data width and datatype of the result may not match that of the source register. See individual instruction descriptions for more details.

## A.1      Opcode Maps

In all of the following opcode maps, cells shaded grey represent reserved opcodes.

### A.1.1 Legacy Opcode Maps

**Primary Opcode Map.** Tables A-1 and A-2 below show the primary opcode map (known in legacy terminology as one-byte opcodes).

Table A-1 below shows those instructions for which the low nibble is in the range 0–7h. Table A-2 on page 452 shows those instructions for which the low nibble is in the range 8–Fh. In both tables, the rows show the full range (0–Fh) of the high nibble, and the columns show the specified range of the low nibble.

**Table A-1. Primary Opcode Map (One-byte Opcodes), Low Nibble 0–7h**

Nibble <sup>1</sup>	0	1	2	3	4	5	6	7
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES <sup>3</sup>	POP ES <sup>3</sup>
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS <sup>3</sup>	POP SS <sup>3</sup>
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg ES <sup>6</sup>	DAA <sup>3</sup>
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg SS <sup>6</sup>	AAA <sup>3</sup>
4	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA <sup>3</sup> PUSHD <sup>3</sup>	POPA <sup>3</sup> POPD <sup>3</sup>	BOUND <sup>3</sup> Gv, Ma	ARPL <sup>3</sup> Ew, Gw MOVSXD <sup>4</sup> Gv, Ez	seg FS prefix	seg GS prefix	operand size override prefix	address size override prefix
7	JO Jb	JNO Jb	JB Jb	JNB Jb	JZ Jb	JNZ Jb	JBE Jb	JNBE Jb
8	Eb, Ib	Ev, Iz	Eb, Ib <sup>3</sup>	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	r8, rAX NOP,PAUSE	rCX/r9, rAX	rDX/r10, rAX	rBX/r11, rAX	rSP/r12, rAX	rBP/r13, rAX	rSI/r14, rAX	rDI/r15, rAX
A	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS B Yb, Xb	MOVSW/D/Q Yv, Xv	CMPSB Xb, Yb	CMPSW/D/Q Xv, Yv
B	AL, Ib r8b, Ib	CL, Ib r9b, Ib	DL, Ib r10b, Ib	BL, Ib r11b, Ib	AH, Ib r12b, Ib	CH, Ib r13b, Ib	DH, Ib r14b, Ib	BH, Ib r15b, Ib
C	Eb, Ib	Ev, Ib	Iw		LES <sup>3</sup> Gz, Mp VEX escape prefix	LDS <sup>3</sup> Gz, Mp VEX escape prefix	Eb, Ib	Ev, Iz
D	Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM Ib <sup>3</sup>	AAD Ib <sup>3</sup>	invalid	XLAT XLATB
E	LOOPNE/NZ Jb	LOOP/Z Jb	LOOP Jb	JrCXZ Jb	IN AL, Ib	eAX, Ib	OUT Ib, AL	Ib, eAX
F	LOCK Prefix	INT1	REPNE Prefix	REP / REPE Prefix	HLT	CMC	Group 3 <sup>2</sup> Eb	Ev

**Notes:**

1. Rows in this table show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal).
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-6 on page 459 for details.
3. Invalid in 64-bit mode.
4. Valid only in 64-bit mode.
5. Used as REX prefixes in 64-bit mode.
6. This is a null prefix in 64-bit mode.

**Table A-2. Primary Opcode Map (One-byte Opcodes), Low Nibble 8–Fh**

Nibble <sup>1</sup>	8	9	A	B	C	D	E	F
0	Eb, Gb	Ev, Gv	Gb, Eb	OR Gv, Ev	AL, Ib	rAX, Iz	PUSH CS <sup>3</sup>	escape to secondary opcode map
1	Eb, Gb	Ev, Gv	Gb, Eb	SBB Gv, Ev	AL, Ib	rAX, Iz	PUSH DS <sup>3</sup>	POP DS <sup>3</sup>
2	Eb, Gb	Ev, Gv	Gb, Eb	SUB Gv, Ev	AL, Ib	rAX, Iz	seg CS <sup>6</sup>	DAS <sup>3</sup>
3	Eb, Gb	Ev, Gv	Gb, Eb	CMP Gv, Ev	AL, Ib	rAX, Iz	seg DS <sup>6</sup>	AAS <sup>3</sup>
4	eAX	eCX	eDX	DEC <sup>3</sup> / REX prefix <sup>5</sup> eBX	eSP	eBP	eSI	eDI
5	rAX/r8	rCX/r9	rDX/r10	POP rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH lz	IMUL Gv, Ev, Iz	PUSH Ib	IMUL Gv, Ev, Ib	INSB Yb, DX	INSW/D Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/D DX, Xz
7	JS Jb	JNS Jb	JP Jb	JNP Jb	JL Jb	JNL Jb	JLE Jb	JNLE Jb
8	Eb, Gb	Ev, Gv	Gb, Eb	MOV Gv, Ev	Mw/Rv, Sw	LEA Gv, M	MOV Sw, Ew	Group 1a <sup>2</sup> XOP escape prefix
9	CBW, CWDE CDQE	CWD, CDQ, CQO	CALL <sup>3</sup> Ap	WAIT FWAIT	PUSHF/D/Q Fv	POPF/D/Q Fv	SAHF	LAHF
A	TEST AL, Ib	STOSB rAX, Iz	STOSW/D/Q Yb, AL	STOSW/D/Q Yv, rAX	LODSB AL, Xb	LODSW/D/Q rAX, Xv	SCASB AL, Yb	SCASW/D/Q rAX, Yv
B	rAX, Iv r8, Iv	rCX, Iv r9, Iv	rDX, Iv r10, Iv	MOV rBX, Iv r11, Iv	rSP, Iv r12, Iv	rBP, Iv r13, Iv	rSI, Iv r14, Iv	rDI, Iv r15, Iv
C	ENTER Iw, Ib	LEAVE	Iw	RET far	INT3	INT Ib	INTO <sup>3</sup>	IRET, IRETD, IRETQ
D	x87 instructions see Table A-15 on page 471							
E	CALL Jz	Jz	JMP Ap <sup>3</sup>	Jb	IN AL, DX eAX, DX	OUT DX, AL DX, eAX		
F	CLC	STC	CLI	STI	CLD	STD	Group 4 <sup>2</sup> Eb	Group 5 <sup>2</sup>

**Notes:**

1. Rows in this table show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal).
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-6 on page 459 for details.
3. Invalid in 64-bit mode.
4. Valid only in 64-bit mode.
5. Used as REX prefixes in 64-bit mode.
6. This is a null prefix in 64-bit mode.

**Secondary Opcode Map.** As described in “Encoding Syntax” on page 1, the escape code 0Fh indicates the switch from the primary to the secondary opcode map. In legacy terminology, the secondary opcode map is presented as a listing of “two-byte” opcodes where the first byte is 0Fh. Tables A-3 and A-4 show the secondary opcode map.

Table A-3 below shows those instructions for which the low nibble is in the range 0–7h. Table A-4 on page 456 shows those instructions for which the low nibble is in the range 8–Fh. In both tables, the rows show the full range (0–Fh) of the high nibble, and the columns show the specified range of the low nibble. Note the added column labeled “prefix.”

For the secondary opcode map shown below, the legacy prefixes 66h, F2h, and F3 are repurposed to provide additional opcode encoding space. For those rows that utilize them, the presence of a 66h, F2h, or F3h prefix changes the operation or the operand types specified by the corresponding opcode value.

As discussed in “Encoding Extensions Using the ModRM Byte” on page 459, some opcode values represent a group of instructions. This is denoted in the map entry by “Group *n*”, where *n* = [1:17,P]. Instructions within a group are encoded by the reg field of the ModRM byte. These encodings are specified in Table A-7 on page 461. For some opcodes, both the reg and the r/m field of the ModRM byte are used to extend the encoding. See Table A-8 on page 462.

**Table A-3. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 0–7h**

Prefix	Nibble <sup>1</sup>	0	1	2	3	4	5	6	7
n/a	0	Group 6 <sup>2</sup>	Group 7 <sup>2</sup>	LAR Gv, Ew	LSL Gv, Ew		SYSCALL	CLTS	SYSRET
none	1	MOVUPS Vps, Wps   Wps, Vps		MOVLPSS Vq, Mq MOVHLPS Vo.q, Uo.q	MOVLPSS Mq, Vq	UNPCKLPS Vps, Wps	UNPCKHPS Vps, Wps	MOVHPS Vo.q, Mq MOVLHPS Vo.q, Uo.q	MOVHPS Mq, Vo.q
		MOVSS Vss, Wss   Wss, Vss		MOVSLDUP Vps, Wps				MOVSHDUP Vps, Wps	
		MOVUPD Vpd, Wpd   Wpd, Vpd		MOVLPD Vo.q, Mq   Mq, Vo.q		UNPCKLPD Vo.q, Wo.q	UNPCKHPD Vo.q, Wo.q	MOVHPD Vo.q, Mq   Mq, Vo.q	
		MOVSD Vsd, Wsd   Wsd, Vsd		MOVDDUP Vo, Wsd					
n/a	2	MOV <sup>4</sup>							
n/a	3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER <sup>3</sup>	SYSEXIT <sup>3</sup>		
n/a	4	CMOVO Gv, Ev	CMOVNO Gv, Ev	CMOVB Gv, Ev	CMOVNB Gv, Ev	CMOVZ Gv, Ev	CMOVNZ Gv, Ev	CMOVBE Gv, Ev	CMOVNBE Gv, Ev
none	5	MOVMSKPS Gd, Ups	SQRTPS Vps, Wps	RSQRTPS Vps, Wps	RCPPS Vps, Wps	ANDPS Vps, Wps	ANDNPS Vps, Wps	ORPS Vps, Wps	XORPS Vps, Wps
F3			SQRTSS Vss, Wss	RSQRTSS Vss, Wss	RCPSS Vss, Wss				
66		MOVMSKPD Gd, Upd	SQRTPD Vpd, Wpd			ANDPD Vpd, Wpd	ANDNPD Vpd, Wpd	ORPD Vpd, Wpd	XORPD Vpd, Wpd
F2			SQRTSD Vsd, Wsd						
none	6	PUN-PCKLBW Pq, Qd	PUN-PCKLWD Pq, Qd	PUN-PCKLDQ Pq, Qd	PACKSSWB Ppi, Qpi	PCMPGTB Ppk, Qpk	PCMPGTW Ppi, Qpi	PCMPGTD Ppj, Qpj	PACKUSWB Ppi, Qpi
F3									
66		PUN-PCKLBW Vo.q, Wo.q	PUN-PCKLWD Vo.q, Wo.q	PUN-PCKLDQ Vo.q, Wo.q	PACKSSWB Vpi, Wpi	PCMPGTB Vpk, Wpk	PCMPGTW Vpi, Wpi	PCMPGTD Vpj, Wpj	PACKUSWB Vpi, Wpi
F2									
none	7	PSHUFW Pq, Qq, Ib	Group 12 <sup>2</sup>	Group 13 <sup>2</sup>	Group 14 <sup>2</sup>	PCMPEQB Ppk, Qpk	PCMPEQW Ppi, Qpi	PCMPEQD Ppj, Qpj	EMMS
F3		PSHUFHW Vq, Wq, Ib							
66		PSHUFD Vo, Wo, Ib				PCMPEQB Vpk, Wpk	PCMPEQW Vpi, Wpi	PCMPEQD Vpj, Wpj	
F2		PSHUFLW Vq, Wq, Ib							

**Notes:**

1. Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceded in the instruction encoding by the escape byte 0Fh.
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 461 for details.
3. Invalid in long mode.
4. Operand size is based on processor mode.

**Table A-3. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 0–7h (continued)**

Prefix	Nibble <sup>1</sup>	0	1	2	3	4	5	6	7
n/a	8	JO Jz	JNO Jz	JB Jz	JNB Jz	JZ Jz	JNZ Jz	JBE Jz	JNBE Jz
n/a	9	SETO Eb	SETNO Eb	SETB Eb	SETNB Eb	SETZ Eb	SETNZ Eb	SETBE Eb	SETNBE Eb
n/a	A	PUSH FS	POP FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib			
n/a	B	CMPXCHG Eb, Gb		LSS Gz, Mp	BTR Ev, Gv	LFS Gz, Mp	LGS Gz, Mp	MOVZX Gv, Eb	
none	C	XADD		CMPPS Vps, Wps, Ib	MOVNTI My, Gy	PINSRW Pq, Ew, Ib	PEXTRW Gd, Nq, Ib	SHUFPS Vps, Wps, Ib	Group 9 <sup>2</sup> Mq
F3		Eb, Gb		CMPSS Vss, Wss, Ib					
66		Ev, Gv		CMPPD Vpd, Wpd, Ib		PINSRW Vo, Ew, Ib	PEXTRW Gd, Uo, Ib	SHUFPD Vpd, Wpd, Ib	
F2		CMPSD Vsd, Wsd, Ib							
none	D		PSRLW Pq, Qq	PSRLD Pq, Qq	PSRLQ Pq, Qq	PADDQ Pq, Qq	PMULLW Pq, Qq		PMOVMSKB Gd, Nq
F3								MOVQ2DQ Vo, Nq	
66		ADDSUBPD Vpd, Wpd	PSRLW Vo, Wo	PSRLD Vo, Wo	PSRLQ Vo, Wo	PADDQ Vo, Wo	PMULLW Vo, Wo	MOVQ Wq, Vq	PMOVMSKB Gd, Uo
F2		ADDSUBPS Vps, Wps						MOVDQ2Q Pq, Uq	
none	E	PAVGB Pq, Qq	PSRAW Pq, Qq	PSRAD Pq, Qq	PAVGW Pq, Qq	PMULHUW Pq, Qq	PMULHW Pq, Qq		MOVNTQ Mq, Pq
F3								CVTDQ2PD Vpd, Wpj	
66		PAVGB Vo, Wo	PSRAW Vo, Wo	PSRAD Vo, Wo	PAVGW Vo, Wo	PMULHUW Vo, Wo	PMULHW Vo, Wo	CVTPD2DQ Vpj, Wpd	MOVNTDQ Mo, Vo
F2								CVTPD2DQ Vpj, Wpd	
none	F		PSLLW Pq, Qq	PSLLD Pq, Qq	PSLLQ Pq, Qq	PMULUDQ Pq, Qq	PMADDWD Pq, Qq	PSADBW Pq, Qq	MASKMOVQ Pq, Nq
F3									
66			PSLLW Vpw, Wo.q	PSLLD Vpwd, Wo.q	PSLLQ Vpqw, Wo.q	PMULUDQ Vpj, Wpj	PMADDWD Vpi, Wpi	PSADBW Vpk, Wpk	MASKMOVDQU Vpb, Upb
F2		LDDQU Vo, Mo							

**Notes:**

1. Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceded in the instruction encoding by the escape byte 0Fh.
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 461 for details.
3. Invalid in long mode.
4. Operand size is based on processor mode.

**Table A-4. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 8–Fh**

Prefix	Nibble <sup>1</sup>	8	9	A	B	C	D	E	F
n/a	0	INVD	WBINVD		UD2		Group P <sup>2</sup> PREFETCH	FEMMS	3DNow! See “3DNow!™ Opcodes” on page 467
n/a	1	Group 16 <sup>2</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>
none	2	MOVAPS Vps, Wps	Wps, Vps	CVTPI2PS Vps, Qpj	MOVNTPS Mo, Vps	CVTTPS2PI Ppj, Wps	CVTPS2PI Ppj, Wps	UCOMISS Vss, Wss	COMISS Vss, Wss
F3				CVTSI2SS Vss, Ey	MOVNTSS Md, Vss	CVTTSS2SI Gy, Wss	CVTSS2SI Gy, Wss		
66		MOVAPD Vpd, Wpd	Wpd, Vpd	CVTPI2PD Vpd, Qpj	MOVNTPD Mo, Vpd	CVTTPD2PI Ppj, Wpd	CVTPD2PI Ppj, Wpd	UCOMISD Vsd, Wsd	COMISD Vsd, Wsd
F2				CVTSI2SD Vsd, Ey	MOVNTSD Mq, Vsd	CVTTSD2SI Gy, Wsd	CVTSD2SI Gy, Wsd		
n/a	3	Escape to 0F_38h opcode map		Escape to 0F_3Ah opcode map					
n/a	4	CMOV\$ Gv, Ev	CMOVNS Gv, Ev	CMOVP Gv, Ev	CMOVNP Gv, Ev	CMOVL Gv, Ev	CMOVNL Gv, Ev	CMOVLE Gv, Ev	CMOVNLE Gv, Ev
none	5	ADDPS Vps, Wps	MULPS Vps, Wps	CVTPS2PD Vpd, Wps	CVTDQ2PS Vps, Wo	SUBPS Vps, Wps	MINPS Vps, Wps	DIVPS Vps, Wps	MAXPS Vps, Wps
F3		ADDSS Vss, Wss	MULSS Vss, Wss	CVTSS2SD Vsd, Wss	CVTTPS2D Q Vo, Wps	SUBSS Vss, Wss	MINSS Vss, Wss	DIVSS Vss, Wss	MAXSS Vss, Wss
66		ADDPD Vpd, Wpd	MULPD Vpd, Wpd	CVTPD2PS Vps, Wpd	CVTPS2DQ Vo, Wps	SUBPD Vpd, Wpd	MINPD Vpd, Wpd	DIVPD Vpd, Wpd	MAXPD Vpd, Wpd
F2		ADDSD Vsd, Wsd	MULSD Vsd, Wsd	CVTSD2SS Vss, Wsd		SUBSD Vsd, Wsd	MINSD Vsd, Wsd	DIVSD Vsd, Wsd	MAXSD Vsd, Wsd
none	6	PUNPCK- HBW Pq, Qd	PUNPCK- HWD Pq, Qd	PUNPCK- HDQ Pq, Qd	PACKSSDW Pq, Qq			MOVD Py, Ey	MOVQ Pq, Qq
F3									MOVDQU Vo, Wo
66		PUNPCK- HBW Vo, Wq	PUNPCK- HWD Vo, Wq	PUNPCK- HDQ Vo, Wq	PACKSSDW Vo, Wo	PUNPCK- LQDQ Vo, Wq	PUNPCK- HQDQ Vo, Wq	MOVD Vy, Ey	MOVDQA Vo, Wo
F2									

**Notes:**

1. Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceded in the instruction encoding by the escape byte 0Fh.
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 461 for details.
3. This instruction takes a ModRM byte.

**Table A-4. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 8–Fh**

Prefix	Nibble <sup>1</sup>	8	9	A	B	C	D	E	F
none	7							MOVD Ey, Py	MOVQ Qq, Pq
F3								MOVQ Vq, Wq	MOVDQU Wo, Vo
66		Group 17 <sup>2</sup> Vo.q, Uo	EXTRQ Vo.q, Uo			HADDPD Vpd, Wpd	HSUBPD Vpd, Wpd	MOVD Ey, Vy	MOVDQA Wo, Vo
F2		INSERTQ Vo.q, Uo.q, Ib, Ib	INSERTQ Vo.q, Uo			HADDPS Vps, Wps	HSUBPS Vps, Wps		
n/a	8	JS Jz	JNS Jz	JP Jz	JNP Jz	JL Jz	JNL Jz	JLE Jz	JNLE Jz
n/a	9	SETS Eb	SETNS Eb	SETP Eb	SETNP Eb	SETL Eb	SETNL Eb	SETLE Eb	SETNLE Eb
n/a	A	PUSH GS	POP GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib		Group 15 <sup>2</sup> Ev, Gv, CL	IMUL Gv, Ev
none	B		Group 10 <sup>2</sup>	Group 8 <sup>2</sup> Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	
F3		POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
F2									
n/a	C	BSWAP							
none	D	PSUBUSB Pq, Qq	PSUBUSW Pq, Qq	PMINUB Pq, Qq	PAND Pq, Qq	PADDUSB Pq, Qq	PADDUSW Pq, Qq	PMAXUB Pq, Qq	PANDN Pq, Qq
F3									
66		PSUBUSB Vo, Wo	PSUBUSW Vo, Wo	PMINUB Vo, Wo	PAND Vo, Wo	PADDUSB Vo, Wo	PADDUSW Vo, Wo	PMAXUB Vo, Wo	PANDN Vo, Wo
F2									
none	E	PSUBSB Pq, Qq	PSUBSW Pq, Qq	PMINSW Pq, Qq	POR Pq, Qq	PADDSB Pq, Qq	PADDSW Pq, Qq	PMAXSW Pq, Qq	PXOR Pq, Qq
F3									
66		PSUBSB Vo, Wo	PSUBSW Vo, Wo	PMINSW Vo, Wo	POR Vo, Wo	PADDSB Vo, Wo	PADDSW Vo, Wo	PMAXSW Vo, Wo	PXOR Vo, Wo
F2									

**Notes:**

1. Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceded in the instruction encoding by the escape byte 0Fh.
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 461 for details.
3. This instruction takes a ModRM byte.

**Table A-4. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 8–Fh**

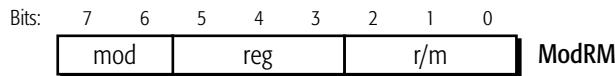
Prefix	Nibble <sup>1</sup>	8	9	A	B	C	D	E	F		
none	F	PSUBB Pq, Qq	PSUBW Pq, Qq	PSUBD Pq, Qq	PSUBQ Pq, Qq	PADDB Pq, Qq	PADDW Pq, Qq	PADDD Pq, Qq	UD0		
F3											
66		PSUBB Vo, Wo	PSUBW Vo, Wo	PSUBD Vo, Wo	PSUBQ Vo, Wo	PADDB Vo, Wo	PADDW Vo, Wo	PADDD Vo, Wo			
F2											
<b>Notes:</b>											
1. Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceded in the instruction encoding by the escape byte 0Fh.											
2. An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 461 for details.											
3. This instruction takes a ModRM byte.											

**rFLAGS Condition Codes for CMOVcc, Jcc, and SETcc Instructions.** Table A-5 shows the rFLAGS condition codes specified by the low nibble in the opcode of the CMOVcc, Jcc, and SETcc instructions.

**Table A-5. rFLAGS Condition Codes for CMOVcc, Jcc, and SETcc**

Low Nibble of Opcode (hex)	rFLAGS Value	cc Mnemonic	Arithmetic Type	Condition(s)
0	OF = 1	O	Signed	Overflow
1	OF = 0	NO		No Overflow
2	CF = 1	B, C, NAE		Below, Carry, Not Above or Equal
3	CF = 0	NB, NC, AE		Not Below, No Carry, Above or Equal
4	ZF = 1	Z, E		Zero, Equal
5	ZF = 0	NZ, NE		Not Zero, Not Equal
6	CF = 1 or ZF = 1	BE, NA		Below or Equal, Not Above
7	CF = 0 and ZF = 0	NBE, A		Not Below or Equal, Above
8	SF = 1	S	Signed	Sign
9	SF = 0	NS		Not Sign
A	PF = 1	P, PE		Parity, Parity Even
B	PF = 0	NP, PO	n/a	Not Parity, Parity Odd
C	(SF xor OF) = 1	L, NGE		Less than, Not Greater than or Equal to
D	(SF xor OF) = 0	NL, GE	Signed	Not Less than, Greater than or Equal to
E	(SF xor OF) = 1 or ZF = 1	LE, NG		Less than or Equal to, Not Greater than
F	(SF xor OF) = 0 and ZF = 0	NLE, G		Not Less than or Equal to, Greater than

**Encoding Extensions Using the ModRM Byte.** The ModRM byte, which immediately follows the opcode byte, is used in certain instruction encodings to provide additional opcode bits with which to define the function of the instruction. ModRM bytes have three fields—*mod*, *reg*, and *r/m*, as shown in Figure A-1.



513-325.eps

**Figure A-1. ModRM-Byte Fields**

In most cases, the *reg* field (bits [5:3]), and in some cases, the *r/m* field (bits [2:0]) provide the additional bits used to extend the encodings of the opcode byte. In the case of the x87 floating-point instructions, the entire ModRM byte is used to extend the opcode encodings.

Table A-6 shows how the ModRM.*reg* field is used to extend the range of opcodes in the primary opcode map. The opcode ranges are organized into *groups* of opcode extensions. The group number is shown in the left-most column. These groups are referenced in the primary opcode map shown in Table A-1 on page 451 and Table A-2 on page 452. An entry of “n.a.” in the Prefix column means that prefixes are not applicable to the opcodes in that row. Prefixes only apply to certain 64-bit media and SSE instructions.

Table A-7 on page 461 shows how the ModRM.*reg* field is used to extend the range of the opcodes in the secondary opcode map.

The /0 through /7 notation for the ModRM *reg* field (bits [5:3]) in the tables below means that the three-bit field contains a value from zero (000b) to 7 (111b).

**Table A-6. ModRM.reg Extensions for the Primary Opcode Map<sup>1</sup>**

Group Number	Prefix	Opcode	ModRM reg Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 1	n/a	80	ADD Eb, Ib	OR Eb, Ib	ADC Eb, Ib	SBB Eb, Ib	AND Eb, Ib	SUB Eb, Ib	XOR Eb, Ib	CMP Eb, Ib
		81	ADD Ev, Iz	OR Ev, Iz	ADC Ev, Iz	SBB Ev, Iz	AND Ev, Iz	SUB Ev, Iz	XOR Ev, Iz	CMP Ev, Iz
		82	ADD Eb, Ib <sup>2</sup>	OR Eb, Ib <sup>2</sup>	ADC Eb, Ib <sup>2</sup>	SBB Eb, Ib <sup>2</sup>	AND Eb, Ib <sup>2</sup>	SUB Eb, Ib <sup>2</sup>	XOR Eb, Ib <sup>2</sup>	CMP Eb, Ib <sup>2</sup>
		83	ADD Ev, Ib	OR Ev, Ib	ADC Ev, Ib	SBB Ev, Ib	AND Ev, Ib	SUB Ev, Ib	XOR Ev, Ib	CMP Ev, Ib

**Notes:**

1. See Table A-7 on page 461 for ModRM extensions for the secondary (two-byte) opcode map.
2. Invalid in 64-bit mode.
3. This instruction takes a ModRM byte.
4. Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.
5. Redundant encoding generally unsupported by tools..

**Table A-6. ModRM.reg Extensions for the Primary Opcode Map<sup>1</sup> (continued)**

Group Number	Prefix	Opcode	ModRM reg Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 1a	n/a	8F	POP Ev							
Group 2	n/a	C0	ROL Eb, Ib	ROR Eb, Ib	RCL Eb, Ib	RCR Eb, Ib	SHL/SAL Eb, Ib	SHR Eb, Ib	SHL/SAL <sup>5</sup> Eb, Ib	SAR Eb, Ib
		C1	ROL Ev, Ib	ROR Ev, Ib	RCL Ev, Ib	RCR Ev, Ib	SHL/SAL Ev, Ib	SHR Ev, Ib	SHL/SAL <sup>5</sup> Ev, Ib	SAR Ev, Ib
		D0	ROL Eb, 1	ROR Eb, 1	RCL Eb, 1	RCR Eb, 1	SHL/SAL Eb, 1	SHR Eb, 1	SHL/SAL <sup>5</sup> Eb, 1	SAR Eb, 1
		D1	ROL Ev, 1	ROR Ev, 1	RCL Ev, 1	RCR Ev, 1	SHL/SAL Ev, 1	SHR Ev, 1	SHL/SAL <sup>5</sup> Ev, 1	SAR Ev, 1
		D2	ROL Eb, CL	ROR Eb, CL	RCL Eb, CL	RCR Eb, CL	SHL/SAL Eb, CL	SHR Eb, CL	SHL/SAL <sup>5</sup> Eb, CL	SAR Eb, CL
		D3	ROL Ev, CL	ROR Ev, CL	RCL Ev, CL	RCR Ev, CL	SHL/SAL Ev, CL	SHR Ev, CL	SHL/SAL <sup>5</sup> Ev, CL	SAR Ev, CL
Group 3	n/a	F6	TEST Eb,lb		NOT Eb	NEG Eb	MUL Eb	IMUL Eb	DIV Eb	IDIV Eb
		F7	TEST Ev,Iz		NOT Ev	NEG Ev	MUL Ev	IMUL Ev	DIV Ev	IDIV Ev
Group 4	n/a	FE	INC Eb	DEC Eb						
Group 5	n/a	FF	INC Ev	DEC Ev	CALL Ev	CALL Mp	JMP Ev	JMP Mp	PUSH Ev	

**Notes:**

1. See Table A-7 on page 461 for ModRM extensions for the secondary (two-byte) opcode map.
2. Invalid in 64-bit mode.
3. This instruction takes a ModRM byte.
4. Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.
5. Redundant encoding generally unsupported by tools..

**Table A-7. ModRM.reg Extensions for the Secondary Opcode Map**

Group Number	Prefix	Opcode	ModRM reg Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 6	n/a	00	SLDT Mw/Rv	STR Mw/Rv	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
Group 7	n/a	01	SGDT Ms	SIDT Ms MONITOR <sup>1</sup> MWAIT	LGDT Ms XGETBV <sup>1</sup> XSETBV	LIDT Ms SVM <sup>1</sup>	SMSW Mw / Rv		LMSW Ew	INVLPG Mb SWAPGS <sup>1</sup> RDTSCP
Group 8	n/a	BA					BT Ev, Ib	BTS Ev, Ib	BTR Ev, Ib	BTC Ev, Ib
Group 9	n/a	C7		CMPXCHG 8B Mq CMPXCHG 16B Mo					RDRAND Rv	
Group 10	n/a	B9	UD1							
Group 11	n/a	C6	MOV Eb,Ib							
	n/a	C7	MOV Ev,Iz							
Group 12	none	71			PSRLW Nq, Ib		PSRAW Nq, Ib		PSLLW Nq, Ib	
	66				PSRLW Uo, Ib		PSRAW Uo, Ib		PSLLW Uo, Ib	
	F2, F3									
Group 13	none	72			PSRLD Nq, Ib		PSRAD Nq, Ib		PSLLD Nq, Ib	
	66				PSRLD Uo, Ib		PSRAD Uo, Ib		PSLLD Uo, Ib	
	F2, F3									
Group 14	none	73			PSRLQ Nq, Ib				PSLLQ Nq, Ib	
	66				PSRLQ Uo, Ib	PSRLDQ Uo, Ib			PSLLQ Uo, Ib	PSLLDQ Uo, Ib
	F2, F3									

**Notes:**

1. Opcode is extended further using the r/m field of the ModRM byte in conjunction with the reg field. See Table A-8 on page 462 for ModRM.r/m extensions of this opcode.
2. Invalid in 64-bit mode.
3. This instruction takes a ModRM byte.
4. Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.
5. ModRM.mod = 11b.
6. ModRM.mod ≠ 11b.
7. ModRM.mod ≠ 11b, ModRM.mod = 11b is an invalid encoding.

**Table A-7. ModRM.reg Extensions for the Secondary Opcode Map (continued)**

Group Number	Prefix	Opcode	ModRM reg Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 15	none	AE	FXSAVE M	FXRSTOR M	LDMXCSR Md	STMXCSR Md	XSAVE M <sup>6</sup>	LFENCE <sup>5</sup> XRSTOR M <sup>6</sup>	MFENCE <sup>5</sup> XSAVE-OPT M <sup>6</sup>	SFENCE <sup>5</sup> CLFLUSH Mb <sup>6</sup>
	F3		RDFSBASE Rv	RDGSSBASE Rv	WRFSBASE Rv	WRGSSBASE Rv				
	66, F2									
Group 16	n/a.	18	PREFETCH NTA	PREFETCH T0	PREFETCH T1	PREFETCH T2	NOP <sup>4</sup>	NOP <sup>4</sup>	NOP <sup>4</sup>	NOP <sup>4</sup>
Group 17	66	78	EXTRQ Vo.q, Ib, Ib							
	none, F2, F3									
Group P	n/a.	0D	PREFETCH Exclusive	PREFETCH Modified	PREFETCH Reserved <sup>4</sup>	PREFETCH Modified	PREFETCH Reserved <sup>4</sup>	PREFETCH Reserved <sup>4</sup>	PREFETCH Reserved <sup>4</sup>	PREFETCH Reserved <sup>4</sup>

**Notes:**

1. Opcode is extended further using the r/m field of the ModRM byte in conjunction with the reg field. See Table A-8 on page 462 for ModRM.r/m extensions of this opcode.
2. Invalid in 64-bit mode.
3. This instruction takes a ModRM byte.
4. Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.
5. ModRM.mod = 11b.
6. ModRM.mod ≠ 11b.
7. ModRM.mod ≠ 11b, ModRM.mod = 11b is an invalid encoding.

**Secondary Opcode Map, ModRM Extensions for Opcode 01h .** Table A-8 below shows the ModRM byte encodings for the 01h opcode. In the table the full ModRM byte is listed below the instruction in hexadecimal. For all instructions shown, the ModRM byte is immediately preceded by the byte string {0Fh, 01h} in the instruction encoding.

**Table A-8. Opcode 01h ModRM Extensions**

reg Field	ModRM.r/m Field							
	0	1	2	3	4	5	6	7
/1	MONITOR (C8)	MWAIT (C9)						
/2	XGETBV (D0)	XSETBV (D1)						
/3	VMRUN (D8)	VMMCALL (D9)	VMLOAD (DA)	VMSAVE (DB)	STGI (DC)	CLGI (DD)	SKINIT (DE)	INVLPGA (DF)
/7	SWAPGS (F8)	RDTSCP (F9)	MONITORX (FA)	MWAITX (FB)				
	ModRM.mod = 11b							

**0F\_38h and 0F\_3Ah Opcode Maps.** The 0F\_38h and 0F\_3Ah opcode maps are used primarily to encode the legacy SSE instructions. In legacy terminology, these maps are presented as three-byte opcodes where the first two bytes are {0Fh, 38h} and {0Fh, 3Ah} respectively.

In these maps the legacy prefixes F2h and F3h are repurposed to provide additional opcode encoding space. In rows [0:E] the legacy prefix 66h is also used to modify the opcode. However, in row F, 66h is used as an operand-size override. See the CRC32 instruction as an example.

The 0F\_38h opcode map is presented below in Tables A-9 and A-10. The 0F\_3Ah opcode map is presented in Tables A-11 and A-12.

**Table A-9. 0F\_38h Opcode Map, Low Nibble = [0h:7h]**

Prefix	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
none	0xh	PSHUFB Ppb, Qpb	PHADDW Ppi, Qpi	PHADDD Ppj, Qpj	PHADDSW Ppi, Qpi	PMADDUBSW Ppk, Qpk	PHSUBW Ppi, Qpi	PHSUBD Ppj, Qpj	PHSUBSW Ppi, Qpi
		PSHUFB Vpb, Wpb	PHADDW Vpi, Wpi	PHADDD Vpj, Wpj	PHADDSW Vpi, Wpi	PMADDUBSW Vpk, Wpk	PHSUBW Vpi, Wpi	PHSUBD Vpj, Wpj	PHSUBSW Vpi, Wpi
66h	1xh								
		PBLENDVB Vpb, Wpb				BLENDVPS Vps, Wps	BLENDVPD Vpd, Wpd		PTEST Vo, Wo
none	2xh								
		PMOVSXBW Vpi, Wpk	PMOVSXBD Vpj, Wpk	PMOVSXBQ Vpq, Wpk	PMOVSXWD Vpj, Wpi	PMOVSXWQ Vpq, Wpi	PMOVSXDDQ Vpq, Wpj		
66h	3xh								
		PMOVZXBW Vpi, Wpk	PMOVZXBD Vpj, Wpk	PMOVZXBQ Vpq, Wpk	PMOVZXWD Vpj, Wpi	PMOVZXWQ Vpq, Wpi	PMOVZXDQ Vpq, Wpj		PCMPGTQ Vpq, Wpq
none	4xh								
		PMULLD Vpj, Wpj	PHMINPOSUW Vpi, Wpi						
...	5xh-Exh	...							
none	Fxh	MOVBE Gv, Mv	MOVBE Mv, Gv						
		CRC32 Gy, Eb	CRC32 Gy, Ev						
		CRC32 Gy, Eb	CRC32 Gy, Ev						

**Table A-10. 0F\_38h Opcode Map, Low Nibble = [8h:Fh]**

Prefix	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
none	0xh	PSIGNB Ppk, Qpk	PSIGNW Ppi, Qpi	PSIGND Ppj, Qpj	PMULHRSW Ppi, Qpi				
		PSIGNB Vpk, Wpk	PSIGNW Vpi, Wpi	PSIGND Vpj, Wpj	PMULHRSW Vpi, Wpi				
none	1xh					PABSB Ppk, Qpk	PABSW Ppi, Qpi	PABSD Ppj, Qpj	
						PABSB Vpk, Wpk	PABSW Vpi, Wpi	PABSD Vpj, Wpj	
none	2xh								
		PMULDQ Vpq, Wpq	PCMPEQQ Vpq, Wpq	MOVNTDQA Vo, Mo	PACKUSDW Vpi, Wpj				
none	3xh								
		PMINSB Vpk, pk	PMINSD Vpj, Wpj	PMINUW Vpi, Wpi	PMINUD Vpj, Wpj	PMAXSB Vpk, Wpk	PMAXSD Vpj, Wpj	PMAXUW Vpi, Wpi	PMAXUD Vpj, Wpj
66h	4xh-Cxh	...							
	Dxh				AESIMC Vo, Wo	AESENC Vo, Wo	AESENCLAST Vo, Wo	AESDEC Vo, Wo	AESDECLAST Vo, Wo
...	Exh-Fxh	...							

**Table A-11. 0F\_3Ah Opcode Map, Low Nibble = [0h:7h]**

Prefix	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
n/a	0xh								
none									
66h	1xh					PEXTRB Mb, Vpk, Ib PEXTRB Ry, Vpk, Ib	PEXTRW Mw, Vpw, Ib PEXTRW Ry, Vpw, Ib	PEXTRD Ed, Vpj, Ib PEXTRQ <sup>1</sup> Eq, Vpq, Ib	EXTRACTPS Md, Vps, Ib EXTRACTPS Ry, Vps, Ib
none	2xh								
66h	PINSRB Vpk, Mb, Ib PINSRB Vpk, Rb, Ib	INSERTPS Vps, Md, Ib INSERTPS Vps, Uo, Ib	PINSRD Vpj, Ed, Ib PINSRQ <sup>1</sup> Vpq, Eq, Ib						
...	3xh	...							
none									
66h	DPPS Vps, Wps, Ib	DPPD Vpd, Wpd, Ib	MPSADBW Vpk, Wpk, Ib		PCLMULQDQ Vpq, Wpq, Ib				
n/a	5xh								
none									
66h	6xh	PCMPESTRM Vo, Wo, Ib	PCMPESTRI Vo, Wo, Ib	PCMPISTRM Vo, Wo, Ib	PCMPISTRI Vo, Wo, Ib				
...	7xh-Exh	...							
n/a	Fxh								
Note 1: When REX prefix is present									

**Table A-12. 0F\_3Ah Opcode Map, Low Nibble = [8h:Fh]**

Prefix	Opcode	x8h	x9h	xAh	xBh	xCb	xDh	xEh	xFh
none	0xh								PALIGNR Ppb, Qpb, Ib
66h	0xh	ROUNDPS Vps, Wps, Ib	ROUNDPD Vpd, Wpd, Ib	ROUNDSS Vss, Wss, Ib	ROUNDSD Vsd, Wsd, Ib	BLENDDPS Vps, Wps, Ib	BLENDDPD Vpd, Wpd, Ib	PBLENDW Vpw, Wpw, Ib	PALIGNR Vpb, Wpb, Ib
...	1xh-Cxh	...							
66h	Dxh								AESKEYGENASSIST Vo, Wo, Ib
...	Fxh	...							

### A.1.2 3DNow!™ Opcodes

The 64-bit media instructions include the MMX™ instructions and the AMD 3DNow!™ instructions. The MMX instructions are encoded using two opcode bytes, as described in “Secondary Opcode Map” on page 452.

The 3DNow! instructions are encoded using two 0Fh opcode bytes and an immediate byte that is located at the last byte position of the instruction encoding. Thus, the format for 3DNow! instructions is:

0Fh 0Fh [ModRM] [SIB] [displacement] *imm8\_opcode*

Table A-13 and Table A-14 on page 469 show the immediate byte following the opcode bytes for 3DNow! instructions. In these tables, rows show the high nibble of the immediate byte, and columns show the low nibble of the immediate byte. Table A-13 shows the immediate bytes whose low nibble is in the range 0–7h. Table A-14 shows the same for immediate bytes whose low nibble is in the range 8–Fh.

Byte values shown as *reserved* in these tables have implementation-specific functions, which can include an invalid-opcode exception.

**Table A-13. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 0–7h**

Nibble <sup>1</sup>	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								
8								
9	PFCMPGE Pq, Qq				PFMIN Pq, Qq		PFRCP Pq, Qq	PFRSQRT Pq, Qq
A	PFCMPGT Pq, Qq				PFMAX Pq, Qq		PFRCPIT1 Pq, Qq	PFRSQIT1 Pq, Qq
B	PFCMPEQ Pq, Qq				PFMUL Pq, Qq		PFRCPIT2 Pq, Qq	PMULHRW Pq, Qq
C								
D								
E								
F								

**Notes:**

1. All 3DNow!™ opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.

**Table A-14. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 8–Fh**

Nibble <sup>1</sup>	8	9	A	B	C	D	E	F
0					PI2FW Pq, Qq	PI2FD Pq, Qq		
1					PF2IW Pq, Qq	PF2ID Pq, Qq		
2								
3								
4								
5								
6								
7								
8			PFNACC Pq, Qq				PFPNACC Pq, Qq	
9			PFSUB Pq, Qq				PFADD Pq, Qq	
A			PFSUBR Pq, Qq				PFACC Pq, Qq	
B				PSWAPD Pq, Qq				PAVGUSB Pq, Qq
C								
D								
E								
F								

**Notes:**

1. All 3DNow!™ opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.

### A.1.3 x87 Encodings

All x87 instructions begin with an opcode byte in the range D8h to DFh, as shown in Table A-2 on page 452. These opcodes are followed by a ModRM byte that further defines the opcode. Table A-15 shows both the opcode byte and the ModRM byte for each x87 instruction.

There are two significant ranges for the ModRM byte for x87 opcodes: 00–BFh and C0–FFh. When the value of the ModRM byte falls within the first range, 00–BFh, the opcode uses only the *reg* field to further define the opcode. When the value of the ModRM byte falls within the second range, C0–FFh, the opcode uses the entire ModRM byte to further define the opcode.

Byte values shown as *reserved* or *invalid* in Table A-15 have implementation-specific functions, which can include an invalid-opcode exception.

The basic instructions FNSTENV, FNSTCW, FNCLEX, FNINIT, FNSAVE, FNSTSW, and FNSTSW do not check for possible floating point exceptions before operating. Utility versions of these mnemonics are provided that insert an FWAIT (opcode 9B) before the corresponding non-waiting instruction. These are FSTENV, FSTCW, FCLEX, FINIT, FSAVE, and FSTSW. For further information on wait and non-waiting versions of these instructions, see their corresponding pages in Volume 5.

Table A-15. x87 Opcodes and ModRM Extensions

Opcode	ModRM <i>mod</i> Field	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
00–BF									
D8	11	FADD mem32rea I	FMUL	FCOM	FCOMP	FSUB	FSUBR mem32rea I	FDIV	FDIVR
		C0 FADD ST(0), ST(0)	C8 FMUL	D0 FCOM	D8 FCOMP	E0 FSUB	E8 FSUBR ST(0), ST(0)	F0 FDIV	F8 FDIVR
		C1 FADD ST(0), ST(1)	C9 FMUL	D1 FCOM	D9 FCOMP	E1 FSUB	E9 FSUBR ST(0), ST(1)	F1 FDIV	F9 FDIVR
		C2 FADD ST(0), ST(2)	CA FMUL	D2 FCOM	DA FCOMP	E2 FSUB	EA FSUBR ST(0), ST(2)	F2 FDIV	FA FDIVR
		C3 FADD ST(0), ST(3)	CB FMUL	D3 FCOM	DB FCOMP	E3 FSUB	EB FSUBR ST(0), ST(3)	F3 FDIV	FB FDIVR
		C4 FADD ST(0), ST(4)	CC FMUL	D4 FCOM	DC FCOMP	E4 FSUB	EC FSUBR ST(0), ST(4)	F4 FDIV	FC FDIVR
		C5 FADD ST(0), ST(5)	CD FMUL	D5 FCOM	DD FCOMP	E5 FSUB	ED FSUBR ST(0), ST(5)	F5 FDIV	FD FDIVR
		C6 FADD ST(0), ST(6)	CE FMUL	D6 FCOM	DE FCOMP	E6 FSUB	EE FSUBR ST(0), ST(6)	F6 FDIV	FE FDIVR
		C7 FADD ST(0), ST(7)	CF FMUL	D7 FCOM	DF FCOMP	E7 FSUB	EF FSUBR ST(0), ST(7)	F7 FDIV	FF FDIVR

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM mod Field	ModRM reg Field								
		/0	/1	/2	/3	/4	/5	/6	/7	
D9	11	!11	FLD mem32rea l		FST mem32real	FSTP mem32real	FLDENV mem14/28en v	FLDCW mem16	FNSTENV mem14/28en v	FNSTCW mem16
		C0	C8	D0	D8	E0	E8	F0	F8	
		FLD ST(0), ST(0)	FXCH ST(0), ST(0)	FNOP	reserved	FCHS	FLD1	F2XM1	FPREM	
		C1	C9	D1	D9	E1	E9	F1	F9	
		FLD ST(0), ST(1)	FXCH ST(0), ST(1)	invalid	reserved	FABS	FLDL2T	FYL2X	FYL2XP1	
		C2	CA	D2	DA	E2	EA	F2	FA	
		FLD ST(0), ST(2)	FXCH ST(0), ST(2)	invalid	reserved	invalid	FLDL2E	FPTAN	FSQRT	
		C3	CB	D3	DB	E3	EB	F3	FB	
		FLD ST(0), ST(3)	FXCH ST(0), ST(3)	invalid	reserved	invalid	FLDP1	FPATAN	FSINCOS	
		C4	CC	D4	DC	E4	EC	F4	FC	
		FLD ST(0), ST(4)	FXCH ST(0), ST(4)	invalid	reserved	FTST	FLDLG2	FXTRACT	FRNDINT	
		C5	CD	D5	DD	E5	ED	F5	FD	
		FLD ST(0), ST(5)	FXCH ST(0), ST(5)	invalid	reserved	FXAM	FLDLN2	FPREM1	FSCALE	
		C6	CE	D6	DE	E6	EE	F6	FE	
		FLD ST(0), ST(6)	FXCH ST(0), ST(6)	invalid	reserved	invalid	FLDZ	FDECSTP	FSIN	
		C7	CF	D7	DF	E7	EF	F7	FF	
		FLD ST(0), ST(7)	FXCH ST(0), ST(7)	invalid	reserved	invalid	FINCSTP	FCOS		

**Table A-15. x87 Opcodes and ModRM Extensions (continued)**

Opcode	ModRM mod Field	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DA	11	00–BF							
		I11 FIADD mem32int	FIMUL mem32int	FICOM mem32int	FICOMP mem32int	FISUB mem32int	FISUBR mem32int	FIDIV mem32int	FIDIVR mem32int
		C0 FCMOVB ST(0), ST(0)	C8 FCMOVE	D0 FCMOVBE	D8 FCMOVU	E0 invalid	E8 invalid	F0 invalid	F8 invalid
		C1 FCMOVB ST(0), ST(1)	C9 FCMOVE	D1 FCMOVBE	D9 FCMOVU	E1 invalid	E9 FUCOMPP	F1 invalid	F9 invalid
		C2 FCMOVB ST(0), ST(2)	CA FCMOVE	D2 FCMOVBE	DA FCMOVU	E2 invalid	EA invalid	F2 invalid	FA invalid
		C3 FCMOVB ST(0), ST(3)	CB FCMOVE	D3 FCMOVBE	DB FCMOVU	E3 invalid	EB invalid	F3 invalid	FB invalid
		C4 FCMOVB ST(0), ST(4)	CC FCMOVE	D4 FCMOVBE	DC FCMOVU	E4 invalid	EC invalid	F4 invalid	FC invalid
		C5 FCMOVB ST(0), ST(5)	CD FCMOVE	D5 FCMOVBE	DD FCMOVU	E5 invalid	ED invalid	F5 invalid	FD invalid
		C6 FCMOVB ST(0), ST(6)	CE FCMOVE	D6 FCMOVBE	DE FCMOVU	E6 invalid	EE invalid	F6 invalid	FE invalid
		C7 FCMOVB ST(0), ST(7)	CF FCMOVE	D7 FCMOVBE	DF FCMOVU	E7 invalid	EF invalid	F7 invalid	FF invalid

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM mod Field	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
00–BF									
DB	!11	FILD mem32int	FISTTP mem32int	FIST mem32int	FISTP mem32int	invalid	FLD mem80rea l	invalid	FSTP mem80real
	11	C0 FCMOVNB ST(0), ST(0)	C8 FCMOVNE ST(0), ST(0)	D0 FCMOVNBE ST(0), ST(0)	D8 FCMOVNU ST(0), ST(0)	E0 reserved	E8 FUCOMI ST(0), ST(0)	F0 FCOMI ST(0), ST(0)	F8 invalid
		C1 FCMOVNB ST(0), ST(1)	C9 FCMOVNE ST(0), ST(1)	D1 FCMOVNBE ST(0), ST(1)	D9 FCMOVNU ST(0), ST(1)	E1 reserved	E9 FUCOMI ST(0), ST(1)	F1 FCOMI ST(0), ST(1)	F9 invalid
		C2 FCMOVNB ST(0), ST(2)	CA FCMOVNE ST(0), ST(2)	D2 FCMOVNBE ST(0), ST(2)	DA FCMOVNU ST(0), ST(2)	E2 FNCLEX	EA FUCOMI ST(0), ST(2)	F2 FCOMI ST(0), ST(2)	FA invalid
		C3 FCMOVNB ST(0), ST(3)	CB FCMOVNE ST(0), ST(3)	D3 FCMOVNBE ST(0), ST(3)	DB FCMOVNU ST(0), ST(3)	E3 FNINIT	EB FUCOMI ST(0), ST(3)	F3 FCOMI ST(0), ST(3)	FB invalid
		C4 FCMOVNB ST(0), ST(4)	CC FCMOVNE ST(0), ST(4)	D4 FCMOVNBE ST(0), ST(4)	DC FCMOVNU ST(0), ST(4)	E4 reserved	EC FUCOMI ST(0), ST(4)	F4 FCOMI ST(0), ST(4)	FC invalid
		C5 FCMOVNB ST(0), ST(5)	CD FCMOVNE ST(0), ST(5)	D5 FCMOVNBE ST(0), ST(5)	DD FCMOVNU ST(0), ST(5)	E5 invalid	ED FUCOMI ST(0), ST(5)	F5 FCOMI ST(0), ST(5)	FD invalid
		C6 FCMOVNB ST(0), ST(6)	CE FCMOVNE ST(0), ST(6)	D6 FCMOVNBE ST(0), ST(6)	DE FCMOVNU ST(0), ST(6)	E6 invalid	EE FUCOMI ST(0), ST(6)	F6 FCOMI ST(0), ST(6)	FE invalid
		C7 FCMOVNB ST(0), ST(7)	CF FCMOVNE ST(0), ST(7)	D7 FCMOVNBE ST(0), ST(7)	DF FCMOVNU ST(0), ST(7)	E7 invalid	EF FUCOMI ST(0), ST(7)	F7 FCOMI ST(0), ST(7)	FF invalid

**Table A-15. x87 Opcodes and ModRM Extensions (continued)**

Opcode	ModRM mod Field	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DC	11	FADD mem64rea l	FMUL mem64real	FCOM mem64real	FCOMP mem64real	FSUB mem64real	FSUBR mem64rea l	FDIV mem64real	FDIVR mem64real
		C0 FADD ST(0), ST(0)	C8 FMUL ST(0), ST(0)	D0 reserved	D8 reserved	E0 FSUBR ST(0), ST(0)	E8 FSUB ST(0), ST(0)	F0 FDIVR ST(0), ST(0)	F8 FDIV ST(0), ST(0)
		C1 FADD ST(1), ST(0)	C9 FMUL ST(1), ST(0)	D1 reserved	D9 reserved	E1 FSUBR ST(1), ST(0)	E9 FSUB ST(1), ST(0)	F1 FDIVR ST(1), ST(0)	F9 FDIV ST(1), ST(0)
		C2 FADD ST(2), ST(0)	CA FMUL ST(2), ST(0)	D2 reserved	DA reserved	E2 FSUBR ST(2), ST(0)	EA FSUB ST(2), ST(0)	F2 FDIVR ST(2), ST(0)	FA FDIV ST(2), ST(0)
		C3 FADD ST(3), ST(0)	CB FMUL ST(3), ST(0)	D3 reserved	DB reserved	E3 FSUBR ST(3), ST(0)	EB FSUB ST(3), ST(0)	F3 FDIVR ST(3), ST(0)	FB FDIV ST(3), ST(0)
		C4 FADD ST(4), ST(0)	CC FMUL ST(4), ST(0)	D4 reserved	DC reserved	E4 FSUBR ST(4), ST(0)	EC FSUB ST(4), ST(0)	F4 FDIVR ST(4), ST(0)	FC FDIV ST(4), ST(0)
		C5 FADD ST(5), ST(0)	CD FMUL ST(5), ST(0)	D5 reserved	DD reserved	E5 FSUBR ST(5), ST(0)	ED FSUB ST(5), ST(0)	F5 FDIVR ST(5), ST(0)	FD FDIV ST(5), ST(0)
		C6 FADD ST(6), ST(0)	CE FMUL ST(6), ST(0)	D6 reserved	DE reserved	E6 FSUBR ST(6), ST(0)	EE FSUB ST(6), ST(0)	F6 FDIVR ST(6), ST(0)	FE FDIV ST(6), ST(0)
		C7 FADD ST(7), ST(0)	CF FMUL ST(7), ST(0)	D7 reserved	DF reserved	E7 FSUBR ST(7), ST(0)	EF FSUB ST(7), ST(0)	F7 FDIVR ST(7), ST(0)	FF FDIV ST(7), ST(0)

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM mod Field	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DD	11	00–BF							
		FLD mem64rea l	FISTTP mem64int	FST mem64real	FSTP mem64real	FRSTOR mem98/108e nv	invalid	FNSAVE mem98/108e nv	FNSTSW mem16
		C0 FFREE ST(0)	C8 reserved	D0 FST ST(0)	D8 FSTP ST(0)	E0 FUCOM ST(0), ST(0)	E8 FUCOMP ST(0)	F0 invalid	F8 invalid
		C1 FFREE ST(1)	C9 reserved	D1 FST ST(1)	D9 FSTP ST(1)	E1 FUCOM ST(1), ST(0)	E9 FUCOMP ST(1)	F1 invalid	F9 invalid
		C2 FFREE ST(2)	CA reserved	D2 FST ST(2)	DA FSTP ST(2)	E2 FUCOM ST(2), ST(0)	EA FUCOMP ST(2)	F2 invalid	FA invalid
		C3 FFREE ST(3)	CB reserved	D3 FST ST(3)	DB FSTP ST(3)	E3 FUCOM ST(3), ST(0)	EB FUCOMP ST(3)	F3 invalid	FB invalid
		C4 FFREE ST(4)	CC reserved	D4 FST ST(4)	DC FSTP ST(4)	E4 FUCOM ST(4), ST(0)	EC FUCOMP ST(4)	F4 invalid	FC invalid
		C5 FFREE ST(5)	CD reserved	D5 FST ST(5)	DD FSTP ST(5)	E5 FUCOM ST(5), ST(0)	ED FUCOMP ST(5)	F5 invalid	FD invalid
		C6 FFREE ST(6)	CE reserved	D6 FST ST(6)	DE FSTP ST(6)	E6 FUCOM ST(6), ST(0)	EE FUCOMP ST(6)	F6 invalid	FE invalid
		C7 FFREE ST(7)	CF reserved	D7 FST ST(7)	DF FSTP ST(7)	E7 FUCOM ST(7), ST(0)	EF FUCOMP ST(7)	F7 invalid	FF invalid

**Table A-15. x87 Opcodes and ModRM Extensions (continued)**

Opcode	ModRM <i>mod</i> Field	ModRM reg Field								
		/0	/1	/2	/3	/4	/5	/6	/7	
DE	11	00–BF								
		!11	FIADD mem16int	FIMUL mem16int	FICOM mem16int	FICOMP mem16int	FISUB mem16int	FISUBR mem16int	FIDIV mem16int	FIDIVR mem16int
		C0 FADDP ST(0), ST(0)	C8 FMULP ST(0), ST(0)	D0 reserved	D8 invalid	E0 FSUBRP ST(0), ST(0)	E8 FSUBP ST(0), ST(0)	F0 FDIVRP ST(0), ST(0)	F8 FDIVP ST(0), ST(0)	
		C1 FADDP ST(1), ST(0)	C9 FMULP ST(1), ST(0)	D1 reserved	D9 FCOMPP	E1 FSUBRP ST(1), ST(0)	E9 FSUBP ST(1), ST(0)	F1 FDIVRP ST(1), ST(0)	F9 FDIVP ST(1), ST(0)	
		C2 FADDP ST(2), ST(0)	CA FMULP ST(2), ST(0)	D2 reserved	DA invalid	E2 FSUBRP ST(2), ST(0)	EA FSUBP ST(2), ST(0)	F2 FDIVRP ST(2), ST(0)	FA FDIVP ST(2), ST(0)	
		C3 FADDP ST(3), ST(0)	CB FMULP ST(3), ST(0)	D3 reserved	DB invalid	E3 FSUBRP ST(3), ST(0)	EB FSUBP ST(3), ST(0)	F3 FDIVRP ST(3), ST(0)	FB FDIVP ST(3), ST(0)	
		C4 FADDP ST(4), ST(0)	CC FMULP ST(4), ST(0)	D4 reserved	DC invalid	E4 FSUBRP ST(4), ST(0)	EC FSUBP ST(4), ST(0)	F4 FDIVRP ST(4), ST(0)	FC FDIVP ST(4), ST(0)	
		C5 FADDP ST(5), ST(0)	CD FMULP ST(5), ST(0)	D5 reserved	DD invalid	E5 FSUBRP ST(5), ST(0)	ED FSUBP ST(5), ST(0)	F5 FDIVRP ST(5), ST(0)	FD FDIVP ST(5), ST(0)	
		C6 FADDP ST(6), ST(0)	CE FMULP ST(6), ST(0)	D6 reserved	DE invalid	E6 FSUBRP ST(6), ST(0)	EE FSUBP ST(6), ST(0)	F6 FDIVRP ST(6), ST(0)	FE FDIVP ST(6), ST(0)	
		C7 FADDP ST(7), ST(0)	CF FMULP ST(7), ST(0)	D7 reserved	DF invalid	E7 FSUBRP ST(7), ST(0)	EF FSUBP ST(7), ST(0)	F7 FDIVRP ST(7), ST(0)	FF FDIVP ST(7), ST(0)	

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM mod Field	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DF	11	00–BF							
		FILD mem16int	FISTTP mem16int	FIST mem16int	FISTP mem16int	FBLD mem80dec	FILD mem64int	FBSTP mem80dec	FISTP mem64int
		C0 reserved	C8 reserved	D0 reserved	D8 reserved	E0 FNSTSW AX	E8 FUCOMIP ST(0), ST(0)	F0 FCOMIP ST(0), ST(0)	F8 invalid
		C1 reserved	C9 reserved	D1 reserved	D9 reserved	E1 invalid	E9 FUCOMIP ST(0), ST(1)	F1 FCOMIP ST(0), ST(1)	F9 invalid
		C2 reserved	CA reserved	D2 reserved	DA reserved	E2 invalid	EA FUCOMIP ST(0), ST(2)	F2 FCOMIP ST(0), ST(2)	FA invalid
		C3 reserved	CB reserved	D3 reserved	DB reserved	E3 invalid	EB FUCOMIP ST(0), ST(3)	F3 FCOMIP ST(0), ST(3)	FB invalid
		C4 reserved	CC reserved	D4 reserved	DC reserved	E4 invalid	EC FUCOMIP ST(0), ST(4)	F4 FCOMIP ST(0), ST(4)	FC invalid
		C5 reserved	CD reserved	D5 reserved	DD reserved	E5 invalid	ED FUCOMIP ST(0), ST(5)	F5 FCOMIP ST(0), ST(5)	FD invalid
		C6 reserved	CE reserved	D6 reserved	DE reserved	E6 invalid	EE FUCOMIP ST(0), ST(6)	F6 FCOMIP ST(0), ST(6)	FE invalid
		C7 reserved	CF reserved	D7 reserved	DF reserved	E7 invalid	EF FUCOMIP ST(0), ST(7)	F7 FCOMIP ST(0), ST(7)	FF invalid

### A.1.4 rFLAGS Condition Codes for x87 Opcodes

Table A-16 shows the rFLAGS condition codes specified by the opcode and ModRM bytes of the FCMOVcc instructions.

**Table A-16. rFLAGS Condition Codes for FCMOVcc**

Opcode (hex)	ModRM <i>mod</i> Field	ModRM <i>reg</i> Field	rFLAGS Value	cc Mnemonic	Condition	
DA	11	000	CF = 1	B	Below	
		001	ZF = 1	E	Equal	
		010	CF = 1 or ZF = 1	BE	Below or Equal	
		011	PF = 1	U	Unordered	
DB		000	CF = 0	NB	Not Below	
		001	ZF = 0	NE	Not Equal	
		010	CF = 0 and ZF = 0	NBE	Not Below or Equal	
		011	PF = 0	NU	Not Unordered	

### A.1.5 Extended Instruction Opcode Maps

The following sections present the VEX and the XOP extended instruction opcode maps. The VEX.map\_select field of the three-byte VEX encoding escape sequence selects VEX opcode maps: 01h, 02h, or 03h. The two-byte VEX encoding escape sequence implicitly selects the VEX map 01h.

The XOP.map\_select field selects between the three XOP maps: 08h, 09h or 0Ah.

**VEX Opcode Maps.** Tables A-17 – A-23 below present the VEX opcode maps and Table A-24 on page 487 presents the VEX opcode groups.

**Table A-17. VEX Opcode Map 1, Low Nibble = [0h:7h]**

Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
00h	...							
1xh	VMOVUPS <sup>2</sup> Vpsx, Wpsx	VMOVUPS <sup>2</sup> Wpsx, Vpsx	VMOVLPS Vps, Hps, Mq VMOVLHPS Vps, Hps, Ups	VMOVLPS Mq, Vps	VUNPCKLPS <sup>2</sup> Vpsx, Hpsx, Wpsx	VUNPCKHPS <sup>2</sup> Vpsx, Hpsx, Wpsx	VMOVHPS Vps, Hps, Mq VMOVLHPS Vps, Hps, Ups	VMOVHPS Mq, Vps
	VMOVUPD <sup>2</sup> Vpdx, Wpdx	VMOVUPD <sup>2</sup> Wpdx, Vpdx	VMOVLPD Vo, Ho, Mq	VMOVLPD Mq, Vo	VUNPCKLPD <sup>2</sup> Vpdx, Hpdx, Wpdx	VUNPCKHPD <sup>2</sup> Vpdx, Hpdx, Wpdx	VMOVHPD Vpd, Hpd, Mq	VMOVHPD Mq, Vpd
	VMOVSS <sup>3</sup> Vss, Md VMOVSS Vss, Hss, Vss	VMOVSS <sup>3</sup> Md, Vss VMOVSS Uss, Hss, Vss	VMOVSLDUP <sup>2</sup> Vpsx, Wpsx				VMOVSHDUP <sup>2</sup> Vpsx, Wpsx	
	VMOVS <sup>3</sup> Vsd, Mq VMOVS Vsd, Hsd, Vsd	VMOVS <sup>3</sup> Mq, Vsd VMOVS Usd, Hsd, Vsd	VMOVDDUP Vo, Wq (L=0) Vdo, Wdo (L=1)					
2xh–4xh	...							
5xh	VMOVMSKPS <sup>2</sup> Gy, Upsx	VSQRTPS <sup>2</sup> Vpsx, Wpsx	VRSQRTPS <sup>2</sup> Vpsx, Wpsx	VRCPPS <sup>2</sup> Vpsx, Wpsx	VANDPS <sup>2</sup> Vpsx, Hpsx, Wpsx	VANDNPS <sup>2</sup> Vpsx, Hpsx, Wpsx	VORPS <sup>2</sup> Vpsx, Hpsx, Wpsx	VXORPS <sup>2</sup> Vpsx, Hpsx, Wpsx
	VMOVMSKPD <sup>2</sup> Gy, Updx	VSQRTPD <sup>2</sup> Vpdx, Wpdx			VANDPD <sup>2</sup> Vpdx, Hpdx, Wpdx	VANDNPD <sup>2</sup> Vpdx, Hpdx, Wpdx	VORPD <sup>2</sup> Vpdx, Hpdx, Wpdx	VXORPD <sup>2</sup> Vpdx, Hpdx, Wpdx
		VSQRTSS <sup>3</sup> Vo, Ho, Wss	VRSQRTSS <sup>3</sup> Vo, Ho, Wss	VRCPPS <sup>3</sup> Vo, Ho, Wss				
		VSQRTSD <sup>3</sup> Vo, Ho, Wsd						
6xh								
	VPUNPCKLBW <sup>2</sup> Vpbx, Hpbx, Wpbx	VPUNPCKLWD <sup>2</sup> Vpx, Hpx, Wpx	VPUNPCKLQ <sup>2</sup> Vpdwx, Hpdwx, Wpdwx	VPACKSSWB <sup>2</sup> Vpkx, Hpix, Wpix	VPCMGTB <sup>2</sup> Vpbx, Hpkx, Wpkx	VPCMGTW <sup>2</sup> Vpx, Hpix, Wpix	VPCMGTD <sup>2</sup> Vpdwx, Hpjk, Wpjx	VPACKUSWB <sup>2</sup> Vpkx, Hpix, Wpix
7xh								VZEROUPPER (L=0) VZEROALL (L=1)
	VPSHUFD <sup>2</sup> Vpdwx, Wpdwx, Ib	VEX group #12	VEX group #13	VEX group #14	VPCMPEQB <sup>2</sup> Vpbx, Hpkx, Wpkx	VPCMPEQW <sup>2</sup> Vpx, Hpix, Wpix	VPCMPEQD <sup>2</sup> Vpdwx, Hpjk, Wpjx	
	VPSHUFHW <sup>2</sup> Vpx, Wpx, Ib							
8xh–Bxh	...							
Cxh			VCMPccPS <sup>1</sup> Vpdw, Hps, Wps, Ib				VSHUFPS <sup>2</sup> Vpsx, Hpsx, Wpsx, Ib	
			VCMPccPD <sup>1</sup> Vpqw, Hpd, Wpd, Ib		VPINSRW Vpw, Hpw, Mw, Ib Vpw, Hpw, Rd, Ib	VPEXTRW Gw, Upw, Ib	VSHUFPD <sup>2</sup> Vpdx, Hpdx, Wpdx, Ib	
			VCMPccSS <sup>1</sup> Vd, Hss, Wss, Ib					
			VCMPccSD <sup>1</sup> Vq, Hsd, Wsd, Ib					

Note 1: The condition codes are: EQ, LT, LE, UNORD, NEQ, NLT, NLE, and ORD; encoded as [00:07h] using Ib.

VEX encoding adds: EQ, UQ, NGE, NGT, FALSE, NEQ, OQ, GE, GT, TRUE [08:0Fh];

EQ\_OS, LT\_OQ, LE\_OQ, UNORD\_S, NEQ\_US, NLT\_UQ, NLE\_UQ, ORD\_S [10h:17h]; and

EQ\_US, NGE\_UQ, NGT\_UQ, FALSE\_OS, NEQ\_OS, GE\_OQ, GT\_OQ, TRUE\_US [18:1Fh].

Note 2: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.

Note 3: Operands are scalars. VEX.L bit is ignored.

**Table A-18. VEX Opcode Map 1, Low Nibble = [0h:7h] Continued**

VEX.pp	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
00	Dxh								
01		VADDSSUBPD <sup>2</sup> Vpdx, Hpdx, Wpdx	VPSRLW <sup>2</sup> Vpxw, Hpxw, Wx	VPSRLD <sup>2</sup> Vpdwx, Hpdwx, Wx	VPSRLQ <sup>2</sup> Vpqwx, Hpqwx, Wx	VPADDQ <sup>2</sup> Vpq, Hpq, Wpq	VPMULLW <sup>2</sup> Vpix, Hpix, Wpix	VMOVQ Wq, Vq (VEX.L=1)	VPMOVMSKB <sup>2</sup> Gy, Upbx
10									
11		VADDSSUBPS <sup>2</sup> Vpsx, Hpsx, Wpsx							
00	Exh								
01		VPAVGB <sup>2</sup> Vpkx, Hpkx, Wpkx	VPSRAW <sup>2</sup> Vpxw, Hpxw, Wx	VPSRAD <sup>2</sup> Vpdwx, Hpdwx, Wx	VPAVGW <sup>2</sup> Vpix, Hpix, Wpix	VPMULHUW <sup>2</sup> Vpi, Hpi, Wpi	VPMULHW Vpi, Hpi, Wpi	VCVTTPD2DQ <sup>2</sup> Vpjx, Wpdx	VMOVNTDQ Mo, Vo (L=0) Mdo, Vdo (L=1)
10								VCVTQDQ2PD <sup>2</sup> Vpdx, Wpjx	
11								VCVTPD2DQ <sup>2</sup> Vpjx, Wpdx	
00	Fxh								
01		VPSLLW <sup>2</sup> Vpxw, Hpxw, Wo.qx	VPSLLD <sup>2</sup> Vpdwx, Hpdwx, Wo.qx	VPSLLQ <sup>2</sup> Vpqwx, Hpqwx, Wo.qx	VPMULUDQ <sup>2</sup> Vpqx, Hpxj, Wpjx	VPMADDWD <sup>2</sup> Vpjx, Hpix, Wpix	VPSADBW <sup>2</sup> Vpix, Hpkk, Wpkx	VMASKMOVDQU Vpb, Upb	
10									
11		VLDDQU Vo, Mo (L=0) Vdo, Mdo (L=1)							
<p>Note 1: The condition codes are: EQ, LT, LE, UNORD, NEQ, NLT, NLE, and ORD; encoded as [00:07h] using lb.      VEX encoding adds: EQ_UQ, NGE, NGT, FALSE, NEQ_OQ, GE, GT, TRUE [08:0Fh];      EQ_OS, LT_OQ, LE_OQ, UNORD_S, NEQ_US, NLT_UQ, NLE_UQ, ORD_S [10h:17h]; and      EQ_US, NGE_UQ, NGT_UQ, FALSE_OS, NEQ_OS, GE_OQ, GT_OQ, TRUE_US [18:1Fh].</p> <p>Note 2: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.</p> <p>Note 3: Operands are scalars. VEX.L bit is ignored.</p>									

**Table A-19. VEX Opcode Map 1, Low Nibble = [8h:Fh]**

VEX.pp	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
...	0xh-1xh	...							
00	2xh	VMOVAPS <sup>1</sup> Vpsx, Wpsx	VMOVAPS <sup>1</sup> Wpsx, Vpsx		VMOVNTPS <sup>1</sup> Mpsx, Vpsx			VUCOMISS <sup>2</sup> Vss, Wss	VCOMISS <sup>2</sup> Vss, Wss
01		VMOVAPD <sup>1</sup> Vpdx, Wpdx	VMOVAPD <sup>1</sup> Wpdx, Vpdx		VMOVNTPD <sup>1</sup> Mpdx, Vpdx			VUCOMISD <sup>2</sup> Vsd, Wsd	VCOMISD <sup>2</sup> Vsd, Wsd
10				VCVTSSI2SS <sup>2</sup> Vo, Ho, Ey		VCVTSS2SI <sup>2</sup> Gy, Wss	VCVTSS2SI <sup>2</sup> Gy, Wss		
11				VCVTSSI2SD <sup>2</sup> Vo, Ho, Ey		VCVTSSD2SI <sup>2</sup> Gy, Wsd	VCVTSSD2SI <sup>2</sup> Gy, Wsd		
...	3xh-4xh	...							
00	5xh	VADDPS <sup>1</sup> Vpsx, Hpsx, Wpsx	VMULPS <sup>1</sup> Vpsx, Hpsx, Wpsx	VCVTPS2PD <sup>1</sup> Vpdx, Wpx	VCVTDQ2PS <sup>1</sup> Vpsx, Wpx	VSUBPS <sup>1</sup> Vpsx, Hpsx, Wpsx	VMINPS <sup>1</sup> Vpsx, Hpsx, Wpsx	VDIVPS <sup>1</sup> Vpsx, Hpsx, Wpsx	VMAXPS <sup>1</sup> Vpsx, Hpsx, Wpsx
01		VADDPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VMULPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VCVTPD2PS <sup>1</sup> Vpsx, Wpdx	VCVTPS2DQ <sup>1</sup> Vpx, Wpsx	VSUBPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VMINPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VDIVPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VMAXPD <sup>1</sup> Vpdx, Hdpx, Wpdx
10		VADDSS <sup>2</sup> Vss, Hss, Wss	VMULSS <sup>2</sup> Vss, Hss, Wss	VCVTSS2SD <sup>2</sup> Vo, Ho, Wss	VCVTPPS2DQ <sup>1</sup> Vpx, Wpsx	VSUBSS <sup>2</sup> Vss, Hss, Wss	VMINSS <sup>2</sup> Vss, Hss, Wss	VDIVSS <sup>2</sup> Vss, Hss, Wss	VMAXSS <sup>2</sup> Vss, Hss, Wss
11		VADDSD <sup>2</sup> Vsd, Hsd, Wsd	VMULSD <sup>2</sup> Vsd, Hsd, Wsd	VCVTSD2SS <sup>2</sup> Vo, Ho, Wsd		VSUBSD <sup>2</sup> Vsd, Hsd, Wsd	VMINSD <sup>2</sup> Vsd, Hsd, Wsd	VDIVSD <sup>2</sup> Vsd, Hsd, Wsd	VMAXSD <sup>2</sup> Vsd, Hsd, Wsd
00	6xh								
01		VPUNPCKHBW <sup>1</sup> Vpbx, Hpbx, Wpbx	VPUNPCKHWD <sup>1</sup> Vpw, Hpw, Wpw	VPUNPCKHDQ <sup>1</sup> Vpdwx, Hpdwx, Wpdwx	VPACKSSDW <sup>1</sup> Vpix, Hpjx, Wpx	VPUNPCKLQDQ <sup>1</sup> Vpqwx, Hpqwx, Wpqwx	VPUNPCKHQDQ <sup>1</sup> Vpqwx, Hpqwx, Wpqwx	VMOVD VMOVQ Vo, Ey (VEX.L=0)	VMOVDQA <sup>1</sup> Vpqwx, Wpqwx
10									VMOVDQU <sup>1</sup> Vpqwx, Wpqwx
11									
00	7xh								
01						VHADDPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VHSUBPD <sup>1</sup> Vpdx, Hdpx, Wpdx	VMOVD VMOVQ Ey, Vo (VEX.L=1)	VMOVDQA <sup>1</sup> Wpqwx, Vpqwx
10								VMOVQ Vq, Wq (VEX.L=0)	VMOVDQU <sup>1</sup> Wpqwx, Vpqwx
11						VHADDP <sup>1</sup> Vpsx, Hpsx, Wpsx	VHSUBP <sup>1</sup> Vpsx, Hpsx, Wpsx		
...	8xh-9xh	...							
n/a	Axh							VEX group #15	
...	Bxh-Cxh	...							
00	Dxh								
01		VPSUBUSB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPSUBUSW <sup>1</sup> Vpix, Hpix, Wpix	VPMINUB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPAND <sup>1</sup> Vx, Hx, Wx	VPADDUSB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPADDUSW <sup>1</sup> Vpix, Hpix, Wpix	VPMAXUB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPAND <sup>1</sup> Vx, Hx, Wx
00	Exh								
01		VPSUBSB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPSUBSW <sup>1</sup> Vpix, Hpix, Wpix	VPMINSW <sup>1</sup> Vpix, Hpix, Wpix	VPOR <sup>1</sup> Vx, Hx, Wx	VPADDSSB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPADDSSW <sup>1</sup> Vpix, Hpix, Wpix	VPMAXSW <sup>1</sup> Vpix, Hpix, Wpix	VPXOR <sup>1</sup> Vx, Hx, Wx
00	Fxh								
01		VPSUBB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPSUBW <sup>1</sup> Vpix, Hpix, Wpix	VPSUBD <sup>1</sup> Vpx, Hpjx, Wpx	VPSUBQ <sup>1</sup> Vpqx, Hpqx, Wpqx	VPADDDB <sup>1</sup> Vpkx, Hpkx, Wpkx	VPADDW <sup>1</sup> Vpix, Hpix, Wpix	VPADD <sup>1</sup> Vpjx, Hpjx, Wpx	
Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits. Note 2: Operands are scalars. VEX.L bit is ignored.									

**Table A-20. VEX Opcode Map 2, Low Nibble = [0h:7h]**

VEX.pp	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
01	0xh	VPSHUFB <sup>1</sup> Vpbx, Hpbx, Wpbx	VPHADDW <sup>1</sup> Vpix, Hpix, Wpix	VPHADDD <sup>1</sup> Vpjx, Hpjx, Wpjx	VPHADDSW <sup>1</sup> Vpix, Hpix, Wpix	VPMADDUBSW <sup>1</sup> Vpix, Hpkx, Wpkx	VPHSUBW <sup>1</sup> Vpix, Hpix, Wpix	VPHSUBD <sup>1</sup> Vpjx, Hpjx, Wpjx	VPHSUBSW <sup>1</sup> Vpix, Hpix, Wpix
01	1xh				VCVTPH2PS <sup>1</sup> Vpsx, Wphx			VPERMPS Vps, Hd, Wps	VPTEST <sup>1,4</sup> Vx, Wx
01	2xh	VPMOVSXBW <sup>1</sup> Vpix, Wpkx	VPMOVSXBD <sup>1</sup> Vpjx, Wpkx	VPMOVSXHQ <sup>1</sup> Vpqx, Wpkx	VPMOVSXWD <sup>1</sup> Vpjx, Wpix	VPMOVSXWQ <sup>1</sup> Vpqx, Wpix	VPMOVSYDQ <sup>1</sup> Vpqx, Wpjx		
01	3xh	VPMOVZXBW <sup>1</sup> Vpix, Wpkx	VPMOVZxbd <sup>1</sup> Vpjx, Wpkx	VPMOVZxbQ <sup>1</sup> Vpqx, Wpkx	VPMOVZXWD <sup>1</sup> Vpjx, Wpix	VPMOVZXWQ <sup>1</sup> Vpqx, Wpix	VPMOVZxDQ <sup>1</sup> Vpqx, Wpjx	VPERMD Vd, Hd, Wd	VPCMPGTQ <sup>1</sup> Vpqx, Hpqx, Wpqx
01	4xh	VPMULLD <sup>1</sup> Vpjx, Hpjx, Wpxj	VPHMINPOSUW Vo, Wpi				VPSRLV- D <sup>1</sup> Vx, Hx, Wx (W=0) Q <sup>1</sup> Vx, Hx, Wx (W=1)	VPSRAVD <sup>1</sup> Vpdwx, Hpdwx, Wpdwx	VPSLLV- D <sup>1</sup> Vx, Hx, Wx (W=0) Q <sup>1</sup> Vx, Hx, Wx (W=1)
...	5xh-8xh	...							
01	9xh	<sup>5</sup> VPGATHERD- D <sup>1</sup> Vx, M*d, Hpdw (W=0) Q <sup>1</sup> Vx, M*q, Hpqwx (W=1)	<sup>5</sup> VPGATHERQ- D <sup>1</sup> Vx, M*d, Hpdw (W=0) Q <sup>1</sup> Vx, M*q, Hpqw (W=1)	<sup>5</sup> VGATHERD- PS <sup>1</sup> Vx,M*ps,Hpsx (W=0)	<sup>5</sup> VGATHERQ- PS <sup>1</sup> Vx,M*ps,Hps (W=0)			<sup>2</sup> VFMADDSSUB132- PS <sup>1</sup> Vx,Hx,Wx (W=0) PD <sup>1</sup> Vx,M*p,HPdx (W=1)	<sup>3</sup> VFMSUBADD132- PS <sup>1</sup> Vx,Hx,Wx (W=0) PD <sup>1</sup> Vx,Hx,Wx (W=1)
01	Axh							VFMADDSSUB213- PS <sup>1</sup> Vx,Hx,Wx (W=0) PD <sup>1</sup> Vx,Hx,Wx (W=1)	VFMSUBADD213- PS <sup>1</sup> Vx,Hx,Wx (W=0) PD <sup>1</sup> Vx,Hx,Wx (W=1)
01	Bxh							VFMADDSSUB231- PS <sup>1</sup> Vx,Hx,Wx (W=0) PD <sup>1</sup> Vx,Hx,Wx (W=1)	VFMSUBADD231- PS <sup>1</sup> Vx,Hx,Wx (W=0) PD <sup>1</sup> Vx,Hx,Wx (W=1)
...	Cxh-Exh	...							
00	Fxh			ANDN Gy, By, Ey	VEX group #17		BZHI Gy, Ey, By		BEXT Gy, Ey, By
01							PEXT Gy, By, Ey		SHLX Gy, Ey, By
10									SARX Gy, Ey, By
11							PDEP Gy, By, Ey	MULX Gy, By, Ey	SHRX Gy, Ey, By
<p>Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.</p> <p>Note 2: For all VFMADDSSUBnnnPS instructions, the data type is packed single-precision floating point. For all VFMADDSSUBnnnPD instructions, the data type is packed double-precision floating point.</p> <p>Note 3: For all VFMSUBADDnnnPS instructions, the data type is packed single-precision floating point. For all VFMSUBADDnnnPD instructions, the data type is packed double-precision floating point.</p> <p>Note 4: Operands are treated as bit vectors.</p> <p>Note 5: Uses VSIB addressing mode.</p>									

**Table A-21. VEX Opcode Map 2, Low Nibble = [8h:Fh]**

VEX.pp	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
01	<b>0xh</b>	VPSIGNB <sup>1</sup> Vpkx, Hpks, Wpkx	VPSIGNW <sup>1</sup> Vpi, Hpi, Wpi	VPSIGND <sup>1</sup> Vpjx, Hpjx, Wpjx	VPMULHRSW <sup>1</sup> Vpix, Hpix, Wpix	VPERMILPS <sup>1</sup> Vpsx, Hpsx, Wpdwx	VPERMILPD <sup>1</sup> Vpdx, Hdpx, Wpqwx	VTESTPS <sup>1</sup> Vpsx, Wpsx	VTESTPD <sup>1</sup> Vpdx, Wpdx
01	<b>1xh</b>	VBROADCASTSS <sup>2</sup> Vps, Wss	VBROADCASTSD Vpd, Wsd (VEX.L=1)	VBROADCASTF128 Vdo, Mo (VEX.L=1)		VPABSB <sup>1</sup> Vpkx, Wpkx	VPABSW <sup>1</sup> Vpix, Wpix	VPABSD <sup>1</sup> Vpjx, Wpjx	
01	<b>2xh</b>	VPMULDQ <sup>1</sup> Vpqx, Hpjx, Wpjx	VPCMPEQQ <sup>1</sup> Vpqx, Hpqx, Wpqx	VMOVNTDQA <sup>1</sup> Vx, Mx	VPACKUSDW <sup>1</sup> Vpix, Hpjx, Wpjx	VMASKMOVPS <sup>1</sup> Vpsx, Hx, Mpsx	VMASKMOVPD <sup>1</sup> Vpdx, Hx, Mpdx	VMASKMOVPS <sup>1</sup> Mpsx, Hx, Vpsx	VMASKMOVPD <sup>1</sup> Mpdx, Hx, Vpdx
01	<b>3xh</b>	VPMINSB <sup>1</sup> Vpkx, Hpks, Wpkx	VPMINSD <sup>1</sup> Vpjx, Hpjx, Wpjx	VPMINUW <sup>1</sup> Vpix, Hpix, Wpix	VPMINUD <sup>1</sup> Vpjx, Hpjx, Wpjx	VPMAXSB <sup>1</sup> Vpkx, Hpks, Wpkx	VPMAXSD <sup>1</sup> Vpjx, Hpjx, Wpjx	VPMAXUW <sup>1</sup> Vpix, Hpix, Wpix	VPMAXUD <sup>1</sup> Vpjx, Hpjx, Wpjx
...	<b>4xh</b>	...							
01	<b>5xh</b>	VPBROADCASTD <sup>1</sup> Vx, Wd	VPBROADCASTQ <sup>1</sup> Vx, Wq	VPBROADCASTI128 Vdo, Mo					
...	<b>6xh</b>	...							
01	<b>7xh</b>	VPBROADCASTB <sup>1</sup> Vx, Wb	VPBROADCASTW <sup>1</sup> Vx, Ww						
01	<b>8xh</b>					VPMASKMOV- D <sup>1</sup> Vx, Hx, Mx (W=0) Q <sup>1</sup> Vx, Hx, Mx (W=1)		VPMASKMOV- D <sup>1</sup> Mx, Hx, Vx (W=0) Q <sup>1</sup> Mx, Hx, Vx (W=1)	
01	<b>9xh</b>	<sup>3</sup> VFMADD132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMADD132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	<sup>4</sup> VFMSUB132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMSUB132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMADD132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMSUB132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMSUB132- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	
01	<b>Axh</b>	VFMADD213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMADD213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMSUB213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMSUB213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMADD213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMSUB213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMSUB213- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	
01	<b>Bxh</b>	VFMADD231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMADD231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMSUB231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFMSUB231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMADD231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMSUB231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	VFNMSUB231- PS <sup>1</sup> Vx, Hx, Wx (W=0) SS <sup>2</sup> Vo, Ho, Wd (W=0) PD <sup>1</sup> Vx, Hx, Wx (W=1) SD <sup>2</sup> Vo, Ho, Wq (W=1)	
...	<b>Cxh</b>	...							
01	<b>Dxh</b>				VAESIMC Vo, Wo	VAESENC Vo, Ho, Wo	VAESENCLAST Vo, Ho, Wo	VAESDEC Vo, Ho, Wo	VAESDECLAST Vo, Ho, Wo
...	<b>Exh-Fxh</b>	...							
<p>Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.</p> <p>Note 2: Operands are scalars. VEX.L bit is ignored.</p> <p>Note 3: For all VFMADDnnnPS instructions, the data type is packed single-precision floating point. For all VFMDADDnnnPD instructions, the data type is packed double-precision floating point.</p> <p>Note 4: For all VFMSUBnnnPS instructions, the data type is packed single-precision floating point. For all VFMSUBnnnPD instructions, the data type is packed double-precision floating point.</p>									

**Table A-22. VEX Opcode Map 3, Low Nibble = [0h:7h]**

VEX.pp	Nibble	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
00	0xh								
01		VPERMQ Vq, Wq, Ib	VPERMPD Vpd, Wpd, Ib	VPBLEND <sup>1</sup> Vpdwx, Hpdwx, Wpdwx, Ib		VPERMILPS <sup>1</sup> Vpsx, Wpsx, Ib	VPERMILPD <sup>1</sup> Vpdx, Wpdx, Ib	VPERM2F128 Vdo, Ho, Wo, Ib (VEX.L=1)	
00	1xh								
01						VPEXTRB Mb, Vpb, Ib VPEXTRB Ry, Vpb, Ib	VPEXTRW Mw, Vpw, Ib VPEXTRW Ry, Vpw, Ib	VPEXTRD Ed, Vpdw, Ib VPEXTRQ Eq, Vpqw, Ib	VEXTRACTPS Mss, Vps, Ib VEXTRACTPS Rss, Vps, Ib
00	2xh								
01		VPINSRB Vpb, Hpb, Wb, Ib	VINSERTPS Vps, Hps, Ups/Md,	VPINSRD Vpdw, Hpdw, Ed, Ib (W=0) VPINSRQ Vpdw, Hpqw, Eq, Ib (W=1)					
...	3xh	...							
00	4xh								
01		VDPPS <sup>1</sup> Vpsx, Hpsx, Wpsx, Ib	VDPPD Vpd, Hpd, Wpd, Ib	VMPSADBW <sup>1</sup> Vpix, Hpx, Wpx, Ib		VPCLMULQDQ Vo, Hpq, Wpq, Ib		VPERM2I128 Vo, Ho, Wo, ib	
...	5xh	...							
00	6xh								
01		VPCMPESTRM Vo, Wo, Ib	VPCMPESTRI Vo, Wo, Ib	VPCMPISTRM Vo, Wo, Ib	VPCMPISTRI Vo, Wo, Ib				
...	7xh-Exh	...							
10	Fxh								
11		RORX Gy, Ey, ib							
Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L=0, size is 128 bits; when L=1, size is 256 bits.									

**Table A-23. VEX Opcode Map 3, Low Nibble = [8h:Fh]**

VEX.pp	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
01	<b>0xh</b>	VROUNDPS <sup>1</sup> Vpsx, Wpsx, Ib	VROUNDPD <sup>1</sup> Vpdx, Wpdx, Ib	VROUNDSS Vss, Hss, Wss, Ib	VROUNDSD Vsd, Hsd, Wsd, Ib	VBLENDPS <sup>1</sup> Vpsx, Hpsx, Wpsx, Ib	VBLENDPD <sup>1</sup> Vpdx, Hpdx, Wpdx, Ib	VPBLENDW <sup>1</sup> Vpx, Hpx, Wpx, Ib	VPALIGNR <sup>1</sup> Vpbx, Hpbx, Wpbx, Ib
01	<b>1xh</b>	VINSERTF128 Vdo, Hdo, Wo, Ib	VEXTRACTF128 Wo, Vdo, Ib				VCVTTPS2PH <sup>1</sup> Wph, Vps, Ib		
...	<b>2xh</b>	...							
01	<b>3xh</b>	VINSERTI128 Vdo, Hdo, Wo, Ib	VEXTRACTI128 Wo, Vdo, Ib						
01	<b>4xh</b>	VPERMILzz2PS <sup>1,2</sup> Vpsx, Hpsx, Wpsx, Lpsx, Ib (W=0) Vpsx, Hpsx, Lpsx, Wpsx, Ib (W=1)	VPERMILzz2PD <sup>1,2</sup> Vpdx, Hpdx, Wpdx, Lpdx, Ib (W=0) Vpdx, Hpdx, Lpdx, Wpdx, Ib (W=1)	VBLENDVPS <sup>1</sup> Vpsx, Hpsx, Wpsx, Lpdx	VBLENDVPD <sup>1</sup> Vpdx, Hpdx, Wpdx, Lpdx	VPBLENDVB <sup>1</sup> Vpbx, Hpbx, Wpbx, Lx			
01	<b>5xh</b>					VFMADDSSUBPS <sup>1</sup> Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMADDSSUBPD <sup>1</sup> Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFMSUBADDP <sup>1</sup> Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMSUBADDPD <sup>1</sup> Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)
01	<b>6xh</b>	VFMADDPS <sup>1</sup> Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMADDPD <sup>1</sup> Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFMADDSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFMADDSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)	VFMSUBPS <sup>1</sup> Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMSUBPD <sup>1</sup> Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFMSUBSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFMSUBSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)
01	<b>7xh</b>	VFNIMADDPS <sup>1</sup> Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFNIMADDPD <sup>1</sup> Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFNIMADDSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFNIMADDSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)	VFNMSUBPS <sup>1</sup> Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFNMSUBPD <sup>1</sup> Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFNMSUBSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFNMSUBSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)
...	<b>8xh-Cxh</b>	...							
01	<b>Dxh</b>								VAESKEYGEN-ASSIST Vo, Wo, Ib
...	<b>Exh-Fxh</b>	...							
Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L=0, size is 128 bits; when L=1, size is 256 bits. Note 2: The zero match codes are TD, TD (alias), MO, and MZ. They are encoded as the zzzz field of the Ib, using 0...3h.									

**Table A-24.** VEX Opcode Groups

Group		ModRM Byte								
Number	VEX Map, Opcode	VEX.pp	xx000xxx	xx001xxx	xx010xxx	xx011xxx	xx100xxx	xx101xxx	xx110xxx	xx111xxx
12	1 71h	01			VPSRLW <sup>1</sup> Hpxw, Upwx, Ib		VPSRAW <sup>1</sup> Hpxw, Upwx, Ib		VPSLLW <sup>1</sup> Hpxw, Upwx, Ib	
13	1 72h	01			VPSRLD <sup>1</sup> Hpdwx, Updwx, Ib		VPSRAD <sup>1</sup> Hpdwx, Updwx, Ib		VPSLLD <sup>1</sup> Hpdwx, Updwx, Ib	
14	1 73h	01			VPSRLQ <sup>1</sup> Hpqwx, Upqwx, Ib	VPSRLDQ <sup>1</sup> Hpbx, Upbx, Ib			VPSLLQ <sup>1</sup> Hpqwx, Upqwx, Ib	VPSLLDQ <sup>1</sup> Hpbx, Upbx, Ib
15	1 AEh	00			VLDMXCSR Md	VSTMXCSR Md				
17	2 F3h	00		BLSR By, Ey	BLSMSK By, Ey	BLSI By, Ey				

Note: 1. Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.

**XOP Opcode Maps.** Tables A-25 – A-30 below present the XOP opcode maps and Table A-31 on page 489 presents the VEX opcode groups.

**Table A-25.** XOP Opcode Map 8h, Low Nibble = [0h:7h]

XOP.pp	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
...	0xh-7xh	...							
00	8xh						VPMACSSWW Vo, Ho, Wo, Lo	VPMACSSWD Vo, Ho, Wo, Lo	VPMACSSDQL Vo, Ho, Wo, Lo
00	9xh						VPMACSWW Vo, Ho, Wo, Lo	VPMACSWD Vo, Ho, Wo, Lo	VPMACSDQL Vo, Ho, Wo, Lo
00	Axh			VPCMOM Vx, Hx, Wx, Lx (W=0) Vx, Hx, Lx, Wx (W=1)	VPPERL Vo, Ho, Wo, Lo (W=0) Vo, Ho, Lo, Wo (W=1)			VPMADCSSWD Vo, Ho, Wo, Lo	
00	Bxh							VPMADCSDW Vo, Ho, Wo, Lo	
00	Cxh	VPROTB Vo, Wo, Ib	VPROTW Vo, Wo, Ib	VPROTD Vo, Wo, Ib	VPROTQ Vo, Wo, Ib				
...	Dxh-Fxh	...							

**Table A-26. XOP Opcode Map 8h, Low Nibble = [8h:Fh]**

XOP.pp	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xF
...	0xh-07h	...							
00	8h						VPMACSSDD Vo,Ho,Wo,Lo	VPMACSSDQH Vo,Ho,Wo,Lo	
00	9h						VPMACSDDD Vo,Ho,Wo,Lo	VPMACSDQH Vo,Ho,Wo,Lo	
...	Axh-Bxh	...							
00	Cxh					VPCOMccB <sup>1</sup> Vo,Ho,Wo,lb	VPCOMccW <sup>1</sup> Vo,Ho,Wo,lb	VPCOMccD <sup>1</sup> Vo,Ho,Wo,lb	VPCOMccQ <sup>1</sup> Vo,Ho,Wo,lb
00	Dxh								
00	Exh					VPCOMccUB <sup>1</sup> Vo,Ho,Wo,lb	VPCOMccUW <sup>1</sup> Vo,Ho,Wo,lb	VPCOMccUD <sup>1</sup> Vo,Ho,Wo,lb	VPCOMccUQ <sup>1</sup> Vo,Ho,Wo,lb
00	Fxh								
Note 1: The condition codes are LT, LE, GT, GE, EQ, NEQ, FALSE, and TRUE. They are encoded via lb, using 00...07h.									

**Table A-27. XOP Opcode Map 9h, Low Nibble = [0h:7h]**

XOP.pp	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
00	0xh		XOP group #1	XOP group #2					
00	1xh			XOP group #3					
...	2xh-7xh	...							
00	8h	VFR CZPS Vx,Wx	VFR CZPD Vx,Wx	VFR CZSS Vq,Wss	VFR CZSD Vq,Wsd				
00	9h	VPROTB Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPROTW Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPROTD Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPROTQ Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLB Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLW Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLD Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLQ Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)
...	Axh-Bxh	...							
00	Cxh		VPHADDWB Vo,Wo	VPHADDBD Vo,Wo	VPHADDBQ Vo,Wo			VPHADDWD Vo,Wo	VPHADDWQ Vo,Wo
00	Dxh		VPHADDUBWD Vo,Wo	VPHADDUBD Vo,Wo	VPHADDUBQ Vo,Wo			VPHADDUWD Vo,Wo	VPHADDUWQ Vo,Wo
00	Exh		VPHSUBBW Vo,Wo	VPHSUBWD Vo,Wo	VPHSUBDQ Vo,Wo				
...	Fxh	...							

**Table A-28. XOP Opcode Map 9h, Low Nibble = [8h:Fh]**

XOP.pp	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xF
...	0xh-8xh	...							
00	9xh	VPSHAB Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHAW Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHAD Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHAQ Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)				
...	Axh-Bxh	...							
00	Cxh				VPHADDDQ Vo,Wo				
00	Dxh				VPHADDUDQ Vo,Wo				
...	Exh-Fxh	...							

**Table A-29. XOP Opcode Map Ah, Low Nibble = [0h:7h]**

XOP.pp	Opcode	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
...	0xh	...							
00	1xh	BEXTR Gy,Ey,Id		XOP group #4					
...	2xh-Fxh	...							

**Table A-30. XOP Opcode Map Ah, Low Nibble = [8h:Fh]**

XOP.pp	Opcode	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
n/a	0xh-Fxh								
Opcodes Reserved									

**Table A-31. XOP Opcode Groups**

ModRM.reg									
Group	/0	/1	/2	/3	/4	/5	/6	/7	
XOP 9 #1 01h		BLCFILL By,Ey	BLSFILL By,Ey	BLCS By,Ey	TZMSK By,Ey	BLCIC By,Ey	BLSIC By,Ey	T1MSKC By,Ey	
XOP 9 #2 02h		BLCMSK By,Ey					BLCI By,Ey		
XOP 9 #3 12h	LLWPBCB Ry	SLWPBCB Ry							
XOP A #4 12h	LWPINS By,Ed,Id	LWPVAL By,Ed,Id							

## A.2 Operand Encodings

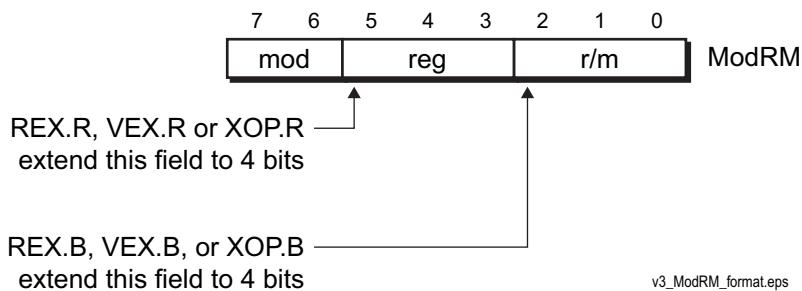
An operand is data that affects or is affected by the execution of an instruction. Operands may be located in registers, memory, or I/O ports. For some instructions, the location of one or more operands is implicitly specified based on the opcode alone. However, for most instructions, operands are specified using bytes that immediately follow the opcode byte. These bytes are designated the *mode-register-memory* (ModRM) byte, the *scale-index-base* (SIB) byte, the displacement byte(s), and the immediate byte(s). The presence of the SIB, displacement, and immediate bytes are optional depending on the instruction, and, for instructions that reference memory, the memory addressing mode.

The following sections describe the encoding of the ModRM and SIB bytes in various processor modes.

### A.2.1 ModRM Operand References

Figure A-2 below shows the format of the ModRM byte. There are three fields—*mod*, *reg*, and *r/m*. The *reg* field is normally used to specify a register-based operand. The *mod* and *r/m* fields together provide a 5-bit field, augmented in 64-bit mode by the R and B bits of a REX, VEX, or XOP prefix, normally used to specify the location of a second memory- or register-based operand and, for a memory-based operand, the addressing mode.

As described in “Encoding Extensions Using the ModRM Byte” on page 459, certain instructions use either the *reg* field, the *r/m* field, or the entire ModRM byte to extend the opcode byte in the encoding of the instruction operation.



**Figure A-2. ModRM-Byte Format**

The two sections below describe the ModRM operand encodings, first for 16-bit references and then for 32-bit and 64-bit references.

**16-Bit Register and Memory References.** Table A-32 shows the notation and encoding conventions for register references using the ModRM *reg* field. This table is comparable to Table A-34 on page 493 but applies only when the address-size is 16-bit. Table A-33 on page 491 shows the

notation and encoding conventions for 16-bit memory references using the ModRM byte. This table is comparable to Table A-35 on page 494.

**Table A-32. ModRM reg Field Encoding, 16-Bit Addressing**

Mnemonic Notation	ModRM reg Field							
	/0	/1	/2	/3	/4	/5	/6	/7
reg8	AL	CL	DL	BL	AH	CH	DH	BH
reg16	AX	CX	DX	BX	SP	BP	SI	DI
reg32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mmx	MMX0	MMX1	MMX2	MMX3	MMX4	MMX5	MMX6	MMX7
xmm	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm	YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sReg	ES	CS	SS	DS	FS	GS	invalid	invalid
cReg	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
dReg	DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7

**Table A-33. ModRM Byte Encoding, 16-Bit Addressing**

Effective Address	ModRM mod Field (binary)	ModRM reg Field <sup>1</sup>								ModRM r/m Field (binary)
		/0	/1	/2	/3	/4	/5	/6	/7	
		Complete ModRM Byte (hex)								
[BX] + [SI]	00	00	08	10	18	20	28	30	38	000
[BX] + [DI]		01	09	11	19	21	29	31	39	001
[BP] + [SI]		02	0A	12	1A	22	2A	32	3A	010
[BP] + [DI]		03	0B	13	1B	23	2B	33	3B	011
[SI]		04	0C	14	1C	24	2C	34	3C	100
[DI]		05	0D	15	1D	25	2D	35	3D	101
disp16		06	0E	16	1E	26	2E	36	3E	110
[BX]		07	0F	17	1F	27	2F	37	3F	111
<b>Notes:</b>										
1. See Table A-32 for complete specification of ModRM “reg” field.										

**Table A-33. ModRM Byte Encoding, 16-Bit Addressing (continued)**

Effective Address	ModRM <i>mod</i> Field (binary)	ModRM reg Field <sup>1</sup>								ModRM <i>r/m</i> Field (binary)
		/0	/1	/2	/3	/4	/5	/6	/7	
		Complete ModRM Byte (hex)								
[BX] + [SI] + disp8	01	40	48	50	58	60	68	70	78	000
[BX] + [DI] + disp8		41	49	51	59	61	69	71	79	001
[BP] + [SI] + disp8		42	4A	52	5A	62	6A	72	7A	010
[BP] + [DI] + disp8		43	4B	53	5B	63	6B	73	7B	011
[SI] + disp8		44	4C	54	5C	64	6C	74	7C	100
[DI] + disp8		45	4D	55	5D	65	6D	75	7D	101
[BP] + disp8		46	4E	56	5E	66	6E	76	7E	110
[BX] + disp8		47	4F	57	5F	67	6F	77	7F	111
[BX] + [SI] + disp16	10	80	88	90	98	A0	A8	B0	B8	000
[BX] + [DI] + disp16		81	89	91	99	A1	A9	B1	B9	001
[BP] + [SI] + disp16		82	8A	92	9A	A2	AA	B2	BA	010
[BP] + [DI] + disp16		83	8B	93	9B	A3	AB	B3	BB	011
[SI] + disp16		84	8C	94	9C	A4	AC	B4	BC	100
[DI] + disp16		85	8D	95	9D	A5	AD	B5	BD	101
[BP] + disp16		86	8E	96	9E	A6	AE	B6	BE	110
[BX] + disp16		87	8F	97	9F	A7	AF	B7	BF	111
AL/ AX/ EAX/ MMX0/ XMM0/ YMM0	11	C0	C8	D0	D8	E0	E8	F0	F8	000
CL/ CX/ ECX/ MMX1/ XMM1/ YMM1		C1	C9	D1	D9	E1	E9	F1	F9	001
DL/ DX/ EDX/ MMX2/ XMM2/ YMM2		C2	CA	D2	DA	E2	EA	F2	FA	010
BL/ BX/ EBX/ MMX3/ XMM3/ YMM3		C3	CB	D3	DB	E3	EB	F3	FB	011
AH/ SP/ ESP/ MMX4/ XMM4/ YMM4		C4	CC	D4	DC	E4	EC	F4	FC	100
CH/ BP/ EBP/ MMX5/ XMM5/ YMM5		C5	CD	D5	DD	E5	ED	F5	FD	101
DH/ SI/ ESI/ MMX6/ XMM6/ YMM6		C6	CE	D6	DE	E6	EE	F6	FE	110
BH/ DI/ EDI/ MMX7/ XMM7/ YMM7		C7	CF	D7	DF	E7	EF	F7	FF	111

**Notes:**

- See Table A-32 for complete specification of ModRM “reg” field.

**Register and Memory References for 32-Bit and 64-Bit Addressing.** Table A-34 on page 493 shows the encoding for register references using the ModRM reg field. The first ten rows of Table A-34 show references when the REX.R bit is cleared to 0, and the last ten rows show references when the REX.R bit is set to 1. In this table, entries under the *Mnemonic Notation* heading correspond

to register notation described in “Mnemonic Syntax” on page 52, and the /r notation under the *ModRM reg Field* heading corresponds to that described in “Opcode Syntax” on page 55.

**Table A-34. ModRM reg Field Encoding, 32-Bit and 64-Bit Addressing**

Mnemonic Notation	REX.R Bit	ModRM reg Field							
		/0	/1	/2	/3	/4	/5	/6	/7
reg8	0	AL	CL	DL	BL	AH/SPL	CH/BPL	DH/SIL	BH/DIL
reg16		AX	CX	DX	BX	SP	BP	SI	DI
reg32		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
reg64		RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
mmx		MMX0	MMX1	MMX2	MMX3	MMX4	MMX5	MMX6	MMX7
xmm		XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm		YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sReg		ES	CS	SS	DS	FS	GS	invalid	invalid
cReg		CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
dReg		DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
reg8	1	R8B	R9B	R10B	R11B	R12B	R13B	R14B	R15B
reg16		R8W	R9W	R10W	R11W	R12W	R13W	R14W	R15W
reg32		R8D	R9D	R10D	R11D	R12D	R13D	R14D	R15D
reg64		R8	R9	R10	R11	R12	R13	R14	R15
mmx		MMX0	MMX1	MMX2	MMX3	MMX4	MMX5	MMX6	MMX7
xmm		XMM8	XMM9	XMM10	XMM11	XMM12	XMM13	XMM14	XMM15
ymm		YMM8	YMM9	YMM10	YMM11	YMM12	YMM13	YMM14	YMM15
sReg		ES	CS	SS	DS	FS	GS	invalid	invalid
cReg		CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
dReg		DR8	DR9	DR10	DR11	DR12	DR13	DR14	DR15

Table A-35 on page 494 shows the encoding for 32-bit and 64-bit memory references using the ModRM byte. This table describes 32-bit and 64-bit addressing, with the REX.B bit set or cleared. The *Effective Address* is shown in the two left-most columns, followed by the binary encoding of the ModRM-byte *mod* field, followed by the eight possible hex values of the complete ModRM byte (one value for each binary encoding of the ModRM-byte *reg* field), followed by the binary encoding of the ModRM *r/m* field.

The /0 through /7 notation for the ModRM *reg* field (bits [5:3]) means that the three-bit field contains a value from zero (binary 000) to 7 (binary 111).

**Table A-35. ModRM Byte Encoding, 32-Bit and 64-Bit Addressing**

Effective Address		ModRM mod Field (binary)	ModRM reg Field <sup>1</sup>								ModRM r/m Field (binary)
			/0	/1	/2	/3	/4	/5	/6	/7	
REX.B = 0	REX.B = 1		Complete ModRM Byte (hex)								
[rAX]	[r8]	00	00	08	10	18	20	28	30	38	000
[rCX]	[r9]		01	09	11	19	21	29	31	39	001
[rDX]	[r10]		02	0A	12	1A	22	2A	32	3A	010
[rBX]	[r11]		03	0B	13	1B	23	2B	33	3B	011
SIB <sup>2</sup>	SIB <sup>2</sup>		04	0C	14	1C	24	2C	34	3C	100
[rIP] + disp32 or disp32 <sup>3</sup>	[rIP] + disp32 or disp32 <sup>3</sup>		05	0D	15	1D	25	2D	35	3D	101
[rSI]	[r14]		06	0E	16	1E	26	2E	36	3E	110
[rDI]	[r15]		07	0F	17	1F	27	2F	37	3F	111
[rAX] + disp8	[r8] + disp8	01	40	48	50	58	60	68	70	78	000
[rCX] + disp8	[r9] + disp8		41	49	51	59	61	69	71	79	001
[rDX] + disp8	[r10] + disp8		42	4A	52	5A	62	6A	72	7A	010
[rBX] + disp8	[r11] + disp8		43	4B	53	5B	63	6B	73	7B	011
[SIB] + disp8	[SIB] + disp8		44	4C	54	5C	64	6C	74	7C	100
[rBP] + disp8	[r13] + disp8		45	4D	55	5D	65	6D	75	7D	101
[rSI] + disp8	[r14] + disp8		46	4E	56	5E	66	6E	76	7E	110
[rDI] + disp8	[r15] + disp8		47	4F	57	5F	67	6F	77	7F	111
[rAX] + disp32	[r8] + disp32	10	80	88	90	98	A0	A8	B0	B8	000
[rCX] + disp32	[r9] + disp32		81	89	91	99	A1	A9	B1	B9	001
[rDX] + disp32	[r10] + disp32		82	8A	92	9A	A2	AA	B2	BA	010
[rBX] + disp32	[r11] + disp32		83	8B	93	9B	A3	AB	B3	BB	011
SIB + disp32	SIB + disp32		84	8C	94	9C	A4	AC	B4	BC	100
[rBP] + disp32	[r13] + disp32		85	8D	95	9D	A5	AD	B5	BD	101
[rSI] + disp32	[r14] + disp32		86	8E	96	9E	A6	AE	B6	BE	110
[rDI] + disp32	[r15] + disp32		87	8F	97	9F	A7	AF	B7	BF	111

**Notes:**

1. See Table A-34 for complete specification of ModRM “reg” field.
2. If SIB.base = 5, the SIB byte is followed by four-byte disp32 field and addressing mode is absolute.
3. In 64-bit mode, the effective address is [rIP]+disp32. In all other modes, the effective address is disp32. If the address-size prefix is used in 64-bit mode to override 64-bit addressing, the [RIP]+disp32 effective address is truncated after computation to 32 bits.

**Table A-35. ModRM Byte Encoding, 32-Bit and 64-Bit Addressing (continued)**

Effective Address		ModRM mod Field (binary)	ModRM reg Field <sup>1</sup>								ModRM r/m Field (binary)
			/0	/1	/2	/3	/4	/5	/6	/7	
REX.B = 0	REX.B = 1	Complete ModRM Byte (hex)									
AL/rAX/MMX0/XMM0/YMM0	r8/MMX0/XMM8/YMM8	11	C0	C8	D0	D8	E0	E8	F0	F8	000
CL/rCX/MMX1/XMM1/YMM1	r9/MMX1/XMM9/YMM9		C1	C9	D1	D9	E1	E9	F1	F9	001
DL/rDX/MMX2/XMM2/YMM2	r10/MMX2/XMM10/YMM10		C2	CA	D2	DA	E2	EA	F2	FA	010
BL/rBX/MMX3/XMM3/YMM3	r11/MMX3/XMM11/YMM11		C3	CB	D3	DB	E3	EB	F3	FB	011
AH/SPL/rSP/MMX4/XMM4/YMM4	r12/MMX4/XMM12/YMM12		C4	CC	D4	DC	E4	EC	F4	FC	100
CH/BPL/rBP/MMX5/XMM5/YMM5	r13/MMX5/XMM13/YMM13		C5	CD	D5	DD	E5	ED	F5	FD	101
DH/SIL/rSI/MMX6/XMM6/YMM6	r14/MMX6/XMM14/YMM14		C6	CE	D6	DE	E6	EE	F6	FE	110
BH/DIL/rDI/MMX7/XMM7/YMM7	r15/MMX7/XMM15/YMM15		C7	CF	D7	DF	E7	EF	F7	FF	111

**Notes:**

1. See Table A-34 for complete specification of ModRM “reg” field.
2. If SIB.base = 5, the SIB byte is followed by four-byte disp32 field and addressing mode is absolute.
3. In 64-bit mode, the effective address is [RIP]+disp32. In all other modes, the effective address is disp32. If the address-size prefix is used in 64-bit mode to override 64-bit addressing, the [RIP]+disp32 effective address is truncated after computation to 32 bits.

## A.2.2 SIB Operand References

Figure A-3 on page 496 shows the format of a scale-index-base (SIB) byte. Some instructions have an SIB byte following their ModRM byte to define memory addressing for the complex-addressing modes described in “Effective Addresses” in Volume 1. The SIB byte has three fields—*scale*, *index*, and *base*—that define the scale factor, index-register number, and base-register number for 32-bit and 64-bit complex addressing modes. In 64-bit mode, the REX.B and REX.X bits extend the encoding of the SIB byte’s *base* and *index* fields.

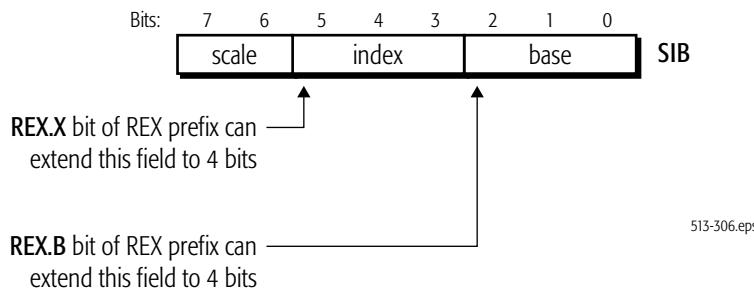
**Figure A-3. SIB Byte Format**

Table A-36 shows the encodings for the SIB byte's *base* field, which specifies the base register for addressing. Table A-37 on page 497 shows the encodings for the effective address referenced by a complete SIB byte, including its *scale* and *index* fields. The /0 through /7 notation for the SIB *base* field means that the three-bit field contains a value between zero (binary 000) and 7 (binary 111).

**Table A-36. Addressing Modes: SIB *base* Field Encoding**

REX.B Bit	ModRM <i>mod</i> Field	SIB <i>base</i> Field							
		/0	/1	/2	/3	/4	/5	/6	/7
0	00	[rAX]	[rCX]	[rDX]	[rBX]	[rSP]	disp32	[rSI]	[rDI]
	01						[rBP] + disp8		
	10						[rBP] + disp32		
1	00	[r8]	[r9]	[r10]	[r11]	[r12]	disp32	[r14]	[r15]
	01						[r13] + disp8		
	10						[r13] + disp32		

**Table A-37. Addressing Modes: SIB Byte Encoding**

Effective Address		SIB scale Field	SIB index Field	SIB base Field <sup>1</sup>								
				REX.B = 0	rAX	rCX	rDX	rBX	rSP	note 1	rSI	rDI
REX.X = 0	REX.X = 1			REX.B = 1	r8	r9	r10	r11	r12	note 1	r14	r15
					/0	/1	/2	/3	/4	/5	/6	/7
Complete SIB Byte (hex)												
[rAX] + [base]	[r8] + [base]	00	000		00	01	02	03	04	05	06	07
[rCX] + [base]	[r9] + [base]		001		08	09	0A	0B	0C	0D	0E	0F
[rDX] + [base]	[r10] + [base]		010		10	11	12	13	14	15	16	17
[rBX] + [base]	[r11] + [base]		011		18	19	1A	1B	1C	1D	1E	1F
[base]	[r12] + [base]		100		20	21	22	23	24	25	26	27
[rBP] + [base]	[r13] + [base]		101		28	29	2A	2B	2C	2D	2E	2F
[rSI] + [base]	[r14] + [base]		110		30	31	32	33	34	35	36	37
[rDI] + [base]	[r15] + [base]		111		38	39	3A	3B	3C	3D	3E	3F
[rAX] * 2 + [base]	[r8] * 2 + [base]	01	000		40	41	42	43	44	45	46	47
[rCX] * 2 + [base]	[r9] * 2 + [base]		001		48	49	4A	4B	4C	4D	4E	4F
[rDX] * 2 + [base]	[r10] * 2 + [base]		010		50	51	52	53	54	55	56	57
[rBX] * 2 + [base]	[r11] * 2 + [base]		011		58	59	5A	5B	5C	5D	5E	5F
[base]	[r12] * 2 + [base]		100		60	61	62	63	64	65	66	67
[rBP] * 2 + [base]	[r13] * 2 + [base]		101		68	69	6A	6B	6C	6D	6E	6F
[rSI] * 2 + [base]	[r14] * 2 + [base]		110		70	71	72	73	74	75	76	77
[rDI] * 2 + [base]	[r15] * 2 + [base]		111		78	79	7A	7B	7C	7D	7E	7F
[rAX] * 4 + [base]	[r8] * 4 + [base]	10	000		80	81	82	83	84	85	86	87
[rCX] * 4 + [base]	[r9] * 4 + [base]		001		88	89	8A	8B	8C	8D	8E	8F
[rDX] * 4 + [base]	[r10] * 4 + [base]		010		90	91	92	93	94	95	96	97
[rBX] * 4 + [base]	[r11] * 4 + [base]		011		98	99	9A	9B	9C	9D	9E	9F
[base]	[r12] * 4 + [base]		100		A0	A1	A2	A3	A4	A5	A6	A7
[rBP]*4+[base]	[r13] * 4 + [base]		101		A8	A9	AA	AB	AC	AD	AE	AF
[rSI]*4+[base]	[r14] * 4 + [base]		110		B0	B1	B2	B3	B4	B5	B6	B7
[rDI]*4+[base]	[r15] * 4 + [base]		111		B8	B9	BA	BB	BC	BD	BE	BF
<b>Notes:</b>												
1. See Table A-36 on page 496 for complete specification of SIB base field.												

**Table A-37. Addressing Modes: SIB Byte Encoding (continued)**

Effective Address		SIB scale Field	SIB index Field	SIB base Field <sup>1</sup>								
				REX.B = 0	rAX	rCX	rDX	rBX	rSP	note 1	rSI	rDI
				REX.B = 1	r8	r9	r10	r11	r12	note 1	r14	r15
REX.X = 0		REX.X = 1			/0	/1	/2	/3	/4	/5	/6	/7
[rAX] * 8 + [base]	[r8] * 8 + [base]	11	000	C0	C1	C2	C3	C4	C5	C6	C7	
[rCX] * 8 + [base]	[r9] * 8 + [base]			C8	C9	CA	CB	CC	CD	CE	CF	
[rDX] * 8 + [base]	[r10] * 8 + [base]			D0	D1	D2	D3	D4	D5	D6	D7	
[rBX] * 8 + [base]	[r11] * 8 + [base]			D8	D9	DA	DB	DC	DD	DE	DF	
[base]	[r12] * 8 + [base]			E0	E1	E2	E3	E4	E5	E6	E7	
[rBP] * 8 + [base]	[r13] * 8 + [base]			E8	E9	EA	EB	EC	ED	EE	EF	
[rSI] * 8 + [base]	[r14] * 8 + [base]			F0	F1	F2	F3	F4	F5	F6	F7	
[rDI] * 8 + [base]	[r15] * 8 + [base]			F8	F9	FA	FB	FC	FD	FE	FF	

**Notes:**

1. See Table A-36 on page 496 for complete specification of SIB base field.

# Appendix B General-Purpose Instructions in 64-Bit Mode

This appendix provides details of the general-purpose instructions in 64-bit mode and its differences from legacy and compatibility modes. The appendix covers only the general-purpose instructions (those described in *Chapter 3, “General-Purpose Instruction Reference”*). It does not cover the 128-bit media, 64-bit media, or x87 floating-point instructions because those instructions are not affected by 64-bit mode, other than in the access by such instructions to extended GPR and XMM registers when using a REX prefix.

## B.1 General Rules for 64-Bit Mode

In 64-bit mode, the following general rules apply to instructions and their operands:

- **“Promoted to 64 Bit”:** If an instruction’s operand size (16-bit or 32-bit) in legacy and compatibility modes depends on the CS.D bit and the operand-size override prefix, then the operand-size choices in 64-bit mode are extended from 16-bit and 32-bit to include 64 bits (with a REX prefix), or the operand size is fixed at 64 bits. Such instructions are said to be “*Promoted to 64 bits*” in Table B-1. However, byte-operand opcodes of such instructions are not promoted.
- **Byte-Operand OpCodes Not Promoted:** As stated above in “Promoted to 64 Bit”, byte-operand opcodes of promoted instructions are not promoted. Those opcodes continue to operate only on bytes.
- **Fixed Operand Size:** If an instruction’s operand size is fixed in legacy mode (thus, independent of CS.D and prefix overrides), that operand size is usually fixed at the same size in 64-bit mode. For example, CPUID operates on 32-bit operands, irrespective of attempts to override the operand size.
- **Default Operand Size:** The default operand size for most instructions is 32 bits, and a REX prefix must be used to change the operand size to 64 bits. However, two groups of instructions default to 64-bit operand size and do not need a REX prefix: (1) near branches and (2) all instructions, except far branches, that implicitly reference the RSP. See Table B-5 on page 527 for a list of all instructions that default to 64-bit operand size.
- **Zero-Extension of 32-Bit Results:** Operations on 32-bit operands in 64-bit mode zero-extend the high 32 bits of 64-bit GPR destination registers.
- **No Extension of 8-Bit and 16-Bit Results:** Operations on 8-bit and 16-bit operands in 64-bit mode leave the high 56 or 48 bits, respectively, of 64-bit GPR destination registers unchanged.
- **Shift and Rotate Counts:** When the operand size is 64 bits, shifts and rotates use one additional bit (6 bits total) to specify shift-count or rotate-count, allowing 64-bit shifts and rotates.
- **Immediates:** The maximum size of immediate operands is 32 bits, except that 64-bit immediates can be MOVED into 64-bit GPRs. Immediates that are less than 64 bits are a maximum of 32 bits, and are sign-extended to 64 bits during use.

- Displacements and Offsets:** The maximum size of an address displacement or offset is 32 bits, except that 64-bit offsets can be used by specific MOV opcodes that read or write AL or rAX. Displacements and offsets that are less than 64 bits are a maximum of 32 bits, and are sign-extended to 64 bits during use.
- Undefined High 32 Bits After Mode Change:** The processor does not preserve the upper 32 bits of the 64-bit GPRs across switches from 64-bit mode to compatibility or legacy modes. In compatibility or legacy mode, the upper 32 bits of the GPRs are undefined and not accessible to software.

## B.2 Operation and Operand Size in 64-Bit Mode

Table B-1 lists the integer instructions, showing operand size in 64-bit mode and the state of the high 32 bits of destination registers when 32-bit operands are used. Opcodes, such as byte-operand versions of several instructions, that do not appear in Table B-1 are covered by the general rules described in “General Rules for 64-Bit Mode” on page 499.

**Table B-1. Operations and Operands in 64-Bit Mode**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>AAA</b> - ASCII Adjust after Addition 37			INVALID IN 64-BIT MODE (invalid-opcode exception)	
<b>AAD</b> - ASCII Adjust AX before Division D5			INVALID IN 64-BIT MODE (invalid-opcode exception)	
<b>AAM</b> - ASCII Adjust AX after Multiply D4			INVALID IN 64-BIT MODE (invalid-opcode exception)	
<b>AAS</b> - ASCII Adjust AL after Subtraction 3F			INVALID IN 64-BIT MODE (invalid-opcode exception)	

**Notes:**

- See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
- The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
- If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
- Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
- Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
- The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>ADC</b> —Add with Carry 11 13 15 81 /2 83 /2	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>ADD</b> —Signed or Unsigned Add 01 03 05 81 /0 83 /0	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>AND</b> —Logical AND 21 23 25 81 /4 83 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>ARPL</b> - Adjust Requestor Privilege Level 63	OPCODE USED as MOVSXD in 64-BIT MODE			
<b>BOUND</b> - Check Array Against Bounds 62	INVALID IN 64-BIT MODE (invalid-opcode exception)			
<b>BSF</b> —Bit Scan Forward 0F BC	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>Notes:</b>				
<ol style="list-style-type: none"> <li>1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.</li> <li>2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.</li> <li>3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</li> <li>4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</li> <li>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</li> <li>6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</li> </ol>				

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>BSR</b> —Bit Scan Reverse 0F BD	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>BSWAP</b> —Byte Swap 0F C8 through 0F CF	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Swap all 8 bytes of a 64-bit GPR.
<b>BT</b> —Bit Test 0F A3 0F BA /4	Promoted to 64 bits.	32 bits	No GPR register results.	
<b>BTC</b> —Bit Test and Complement 0F BB 0F BA /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>BTR</b> —Bit Test and Reset 0F B3 0F BA /6	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>BTS</b> —Bit Test and Set 0F AB 0F BA /5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>CALL</b> —Procedure Call Near	See “Near Branches in 64-Bit Mode” in Volume 1.			
E8	Promoted to 64 bits.	64 bits	Can’t encode. <sup>6</sup>	RIP = RIP + 32-bit displacement sign-extended to 64 bits.
FF /2	Promoted to 64 bits.	64 bits	Can’t encode. <sup>6</sup>	RIP = 64-bit offset from register or memory.
<b>Notes:</b>				
1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.				
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.				
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.				
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.				
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.				
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.				

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>		
<b>CALL</b> —Procedure Call Far  9A  FF /3	See “Branches to 64-Bit Offsets” in Volume 1.					
	INVALID IN 64-BIT MODE (invalid-opcode exception)					
	Promoted to 64 bits.	32 bits	If selector points to a gate, then RIP = 64-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction.			
<b>CBW, CWDE, CDQE</b> —Convert Byte to Word, Convert Word to Doubleword, Convert Doubleword to Quadword  98	Promoted to 64 bits.	32 bits (size of destination register)	CWDE: Converts word to doubleword. Zero-extends EAX to RAX.  RAX = sign-extended EAX.	CDQE (new mnemonic): Converts doubleword to quadword.		
<b>CDQ</b>	see <b>CWD, CDQ, CQO</b>					
<b>CDQE</b> (new mnemonic)	see <b>CBW, CWDE, CDQE</b>					
<b>CDWE</b>	see <b>CBW, CWDE, CDQE</b>					
<b>CLC</b> —Clear Carry Flag  F8	Same as legacy mode.	Not relevant.	No GPR register results.			
<b>CLD</b> —Clear Direction Flag  FC	Same as legacy mode.	Not relevant.	No GPR register results.			
<b>CLFLUSH</b> —Cache Line Invalidate  0F AE /7	Same as legacy mode.	Not relevant.	No GPR register results.			
<b>CLGI</b> —Clear Global Interrupt  0F 01 DD	Same as legacy mode	Not relevant	No GPR register results.			
<b>CLI</b> —Clear Interrupt Flag  FA	Same as legacy mode.	Not relevant.	No GPR register results.			
<b>Notes:</b>						
<ol style="list-style-type: none"> <li>1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.</li> <li>2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.</li> <li>3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</li> <li>4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.</li> <li>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</li> <li>6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</li> </ol>						

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>CLTS</b> —Clear Task-Switched Flag in CR0 0F 06	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>CMC</b> —Complement Carry Flag F5	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>CMOVcc</b> —Conditional Move 0F 40 through 0F 4F	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits. This occurs even if the condition is false.	
<b>CMP</b> —Compare 39 3B 3D 81 /7 83 /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>CMPS, CMPSW, CMPSD, CMPSQ</b> —Compare Strings A7	Promoted to 64 bits.	32 bits	CMPSD: Compare String Doublewords. See footnote <sup>5</sup>	CMPSQ (new mnemonic): Compare String Quadwords See footnote <sup>5</sup>
<b>CMPXCHG</b> —Compare and Exchange 0F B1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>				
<b>CMPXCHG8B</b> —Compare and Exchange Eight Bytes  0F C7 /1	Same as legacy mode.	32 bits.	Zero-extends EDX and EAX to 64 bits.	CMPXCHG16B (new mnemonic): Compare and Exchange 16 Bytes.				
<b>CPUID</b> —Processor Identification  0F A2	Same as legacy mode.	Operand size fixed at 32 bits.	Zero-extends 32-bit register results to 64 bits.					
<b>CQO</b> (new mnemonic)	see <b>CWD, CDQ, CQO</b>							
<b>CWD, CDQ, CQO</b> —Convert Word to Doubleword, Convert Doubleword to Quadword, Convert Quadword to Double Quadword  99	Promoted to 64 bits.	32 bits (size of destination register)	CDQ: Converts doubleword to quadword. Sign-extends EAX to EDX. Zero-extends EDX to RDX. RAX is unchanged.	CQO (new mnemonic): Converts quadword to double quadword. Sign-extends RAX to RDX. RAX is unchanged.				
<b>DAA</b> - Decimal Adjust AL after Addition 27	INVALID IN 64-BIT MODE (invalid-opcode exception)							
<b>DAS</b> - Decimal Adjust AL after Subtraction 2F	INVALID IN 64-BIT MODE (invalid-opcode exception)							
<b>Notes:</b>								
<ol style="list-style-type: none"> <li>1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.</li> <li>2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.</li> <li>3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</li> <li>4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.</li> <li>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</li> <li>6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</li> </ol>								

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>DEC</b> —Decrement by 1 FF /1  48 through 4F	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
	OPCODE USED as REX PREFIX in 64-BIT MODE			
<b>DIV</b> —Unsigned Divide  F7 /6	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX contain a 64-bit quotient (RAX) and 64-bit remainder (RDX).
<b>ENTER</b> —Create Procedure Stack Frame  C8	Promoted to 64 bits.	64 bits	Can't encode <sup>6</sup>	
<b>HLT</b> —Halt  F4	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>IDIV</b> —Signed Divide  F7 /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX contain a 64-bit quotient (RAX) and 64-bit remainder (RDX).

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>IMUL</b> - Signed Multiply	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX = RAX * reg/mem64 (i.e., 128-bit result)
F7 /5				reg64 = reg64 * reg/mem64
0F AF				reg64 = reg/mem64 * imm32
69				reg64 = reg/mem64 * imm8
6B				
<b>IN</b> —Input From Port	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
E5				
ED				
<b>INC</b> —Increment by 1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
FF /0				
40 through 47			OPCODE USED as REX PREFIX in 64-BIT MODE	
<b>INS, INSW, INSD</b> —Input String	Same as legacy mode.	32 bits	INSD: Input String Doublewords. No GPR register results. See footnote <sup>5</sup>	
6D				
<b>INT n</b> —Interrupt to Vector	Promoted to 64 bits.	Not relevant.	See “Long-Mode Interrupt Control Transfers” in Volume 2.	
CD				
<b>INT3</b> —Interrupt to Debug Vector				
CC				
<b>INTO</b> - Interrupt to Overflow Vector	INVALID IN 64-BIT MODE (invalid-opcode exception)			
CE				

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>INVD</b> —Invalidate Internal Caches 0F 08	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>INVLPG</b> —Invalidate TLB Entry 0F 01 /7	Promoted to 64 bits.	Not relevant.	No GPR register results.	
<b>INVLPGA</b> —Invalidate TLB Entry in a Specified ASID	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>IRET, IRETD, IRETQ</b> —Interrupt Return  CF	Promoted to 64 bits.	32 bits	IRETD: Interrupt Return Doubleword. See “Long-Mode Interrupt Control Transfers” in Volume 2.	IRETQ (new mnemonic): Interrupt Return Quadword. See “Long-Mode Interrupt Control Transfers” in Volume 2.
<b>Jcc</b> —Jump Conditional  70 through 7F  0F 80 through 0F 8F	Promoted to 64 bits.	64 bits	See “Near Branches in 64-Bit Mode” in Volume 1.  Can’t encode. <sup>6</sup>	RIP = RIP + 8-bit displacement sign-extended to 64 bits.
				RIP = RIP + 32- bit displacement sign-extended to 64 bits.
				RIP = RIP + 8-bit displacement sign-extended to 64 bits. See footnote <sup>5</sup>
<b>JCXZ, JECXZ, JRCXZ</b> —Jump on CX/ECX/RCX Zero  E3	Promoted to 64 bits.	64 bits	Can’t encode. <sup>6</sup>	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>JMP</b> —Jump Near	See “Near Branches in 64-Bit Mode” in Volume 1.			
EB				RIP = RIP + 8-bit displacement sign-extended to 64 bits.
E9	Promoted to 64 bits.	64 bits	Can't encode. <sup>6</sup>	RIP = RIP + 32-bit displacement sign-extended to 64 bits.
FF /4				RIP = 64-bit offset from register or memory.
<b>JMP</b> —Jump Far	See “Branches to 64-Bit Offsets” in Volume 1.			
EA	INVALID IN 64-BIT MODE (invalid-opcode exception)			
FF /5	Promoted to 64 bits.	32 bits	If selector points to a gate, then RIP = 64-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction.	
<b>LAHF</b> - Load Status Flags into AH Register	Same as legacy mode.	Not relevant.		
9F				
<b>LAR</b> —Load Access Rights Byte	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
0F 02				
<b>LDS</b> - Load DS Far Pointer	INVALID IN 64-BIT MODE (invalid-opcode exception)			
C5				

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>LEA</b> —Load Effective Address 8D	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>LEAVE</b> —Delete Procedure Stack Frame C9	Promoted to 64 bits.	64 bits	Can't encode <sup>6</sup>	
<b>LES</b> - Load ES Far Pointer C4	INVALID IN 64-BIT MODE (invalid-opcode exception)			
<b>LFENCE</b> —Load Fence 0F AE /5	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>LFS</b> —Load FS Far Pointer 0F B4	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>LGDT</b> —Load Global Descriptor Table Register 0F 01 /2	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Loads 8-byte base and 2-byte limit.	
<b>LGS</b> —Load GS Far Pointer 0F B5	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>LIDT</b> —Load Interrupt Descriptor Table Register 0F 01 /3	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Loads 8-byte base and 2-byte limit.	
<b>LLDT</b> —Load Local Descriptor Table Register 0F 00 /2	Promoted to 64 bits.	Operand size fixed at 16 bits.	No GPR register results. References 16-byte descriptor to load 64-bit base.	
<b>LMSW</b> —Load Machine Status Word 0F 01 /6	Same as legacy mode.	Operand size fixed at 16 bits.	No GPR register results.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>LODS, LODSW, LODSD, LODSQ</b> — Load String  AD	Promoted to 64 bits.	32 bits	LODSD: Load String Doublewords. Zero-extends 32- bit register results to 64 bits. See footnote <sup>5</sup>	LODSQ (new mnemonic): Load String Quadwords. See footnote <sup>5</sup>
<b>LOOP</b> —Loop E2	Promoted to 64 bits.	64 bits	Can't encode. <sup>6</sup>	RIP = RIP + 8-bit displacement sign-extended to 64 bits. See footnote <sup>5</sup>
<b>LOOPZ, LOOPE</b> —Loop if Zero/Equal E1				
<b>LOOPNZ, LOOPNE</b> —Loop if Not Zero/Equal  E0				
<b>LSL</b> —Load Segment Limit 0F 03	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>LSS</b> —Load SS Segment Register 0F B2	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>LTR</b> —Load Task Register 0F 00 /3	Promoted to 64 bits.	Operand size fixed at 16 bits.	No GPR register results. References 16-byte descriptor to load 64-bit base.	
<b>LZCNT</b> —Count Leading Zeros F3 0F BD	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>MFENCE</b> —Memory Fence 0F AE /6	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>MONITOR</b> —Setup Monitor Address 0F 01 C8	Same as legacy mode.	Operand size fixed at 32 bits.	No GPR register results.	
<b>Notes:</b>				
<ol style="list-style-type: none"> <li>1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.</li> <li>2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.</li> <li>3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</li> <li>4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.</li> <li>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</li> <li>6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</li> </ol>				

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>				
<b>MOV</b> —Move	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.					
89				32-bit immediate is sign-extended to 64 bits.				
8B				64-bit immediate.				
C7								
B8 through BF A1 (moffset)			Zero-extends 32-bit register results to 64 bits. Memory offsets are address-sized and default to 64 bits.	Memory offsets are address-sized and default to 64 bits.				
A3 (moffset)								
<b>MOV</b> —Move to/from Segment Registers	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.					
8C								
8E			Operand size fixed at 16 bits.	No GPR register results.				
<b>MOV(CRn)</b> —Move to/from Control Registers	Promoted to 64 bits.	Operand size fixed at 64 bits.	The high 32 bits of control registers differ in their writability and reserved status. See “System Resources” in Volume 2 for details.					
0F 22								
0F 20								
<b>MOV(DRn)</b> —Move to/from Debug Registers	Promoted to 64 bits.	Operand size fixed at 64 bits.	The high 32 bits of debug registers differ in their writability and reserved status. See “Debug and Performance Resources” in Volume 2 for details.					
0F 21								
0F 23								
<b>Notes:</b>								
1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.								
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.								
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.								
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.								
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.								
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.								

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>		
<b>MOVD</b> —Move Doubleword or Quadword  0F 6E 0F 7E 66 0F 6E 66 0F 7E	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.			
			Zero-extends 32-bit register results to 128 bits.	Zero-extends 64-bit register results to 128 bits.		
<b>MOVNTI</b> —Move Non-Temporal Doubleword  0F C3	Promoted to 64 bits.	32 bits	No GPR register results.			
<b>MOVS, MOVSW, MOVSQ, MOVSD</b> —Move String  A5	Promoted to 64 bits.	32 bits	MOVSD: Move String Doublewords. See footnote <sup>5</sup>	MOVSQ (new mnemonic): Move String Quadwords. See footnote <sup>5</sup>		
<b>MOVSX</b> —Move with Sign-Extend  0F BE  0F BF	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Sign-extends byte to quadword.		
				Sign-extends word to quadword.		
<b>Notes:</b>						
1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.						
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.						
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.						
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.						
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.						
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.						

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>MOVSXD</b> —Move with Sign-Extend Doubleword  63	New instruction, available only in 64-bit mode. (In other modes, this opcode is ARPL instruction.)	32 bits	Zero-extends 32-bit register results to 64 bits.	Sign-extends doubleword to quadword.
<b>MOVZX</b> —Move with Zero-Extend  0F B6  0F B7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Zero-extends byte to quadword.
				Zero-extends word to quadword.
<b>MUL</b> —Multiply Unsigned  F7 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX=RAX* quadword in register or memory.
<b>MWAIT</b> —Monitor Wait  0F 01 C9	Same as legacy mode.	Operand size fixed at 32 bits.	No GPR register results.	
<b>NEG</b> —Negate Two's Complement  F7 /3	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>NOP</b> —No Operation  90	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>NOT</b> —Negate One's Complement  F7 /2	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>OR</b> —Logical OR 09 0B 0D 81 /1 83 /1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>OUT</b> —Output to Port E7 EF	Same as legacy mode.	32 bits	No GPR register results.	
<b>OUTS, OUTSW, OUTSD</b> —Output String 6F	Same as legacy mode.	32 bits	Writes doubleword to I/O port. No GPR register results. See footnote <sup>5</sup>	
<b>PAUSE</b> —Pause F3 90	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>POP</b> —Pop Stack 8F /0 58 through 5F	Promoted to 64 bits.	64 bits	Cannot encode <sup>6</sup>	No GPR register results.
<b>POP</b> —Pop (segment register from) Stack 0F A1 (POP FS) 0F A9 (POP GS) 1F (POP DS) 07 (POP ES) 17 (POP SS)	Same as legacy mode.	64 bits	Cannot encode <sup>6</sup>	No GPR register results.
			INVALID IN 64-BIT MODE (invalid-opcode exception)	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>POPA, POPAD</b> —Pop All to GPR Words or Doublewords 61	INVALID IN 64-BIT MODE (invalid-opcode exception)			
<b>POPCNT</b> —Bit Population Count F3 0F B8	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>POPF, POPFD, POPFQ</b> —Pop to rFLAGS Word, Doublword, or Quadword 9D	Promoted to 64 bits.	64 bits	Cannot encode <sup>6</sup>	POPFQ (new mnemonic): Pops 64 bits off stack, writes low 32 bits into EFLAGS and zero-extends the high 32 bits of RFLAGS.
<b>PREFETCH</b> —Prefetch L1 Data-Cache Line 0F 0D /0	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>PREFETCH/level</b> —Prefetch Data to Cache Level /level 0F 18 /0-3	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>PREFETCHW</b> —Prefetch L1 Data-Cache Line for Write 0F 0D /1	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>PUSH</b> —Push onto Stack FF /6 50 through 57 6A 68	Promoted to 64 bits.	64 bits	Cannot encode <sup>6</sup>	
<b>Notes:</b>				
1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.				
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.				
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.				
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.				
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.				
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.				

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>PUSH</b> —Push (segment register) onto Stack 0F A0 (PUSH FS) 0F A8 (PUSH GS) 0E (PUSH CS) 1E (PUSH DS) 06 (PUSH ES) 16 (PUSH SS)	Promoted to 64 bits.	64 bits	Cannot encode <sup>6</sup>	
			INVALID IN 64-BIT MODE (invalid-opcode exception)	
<b>PUSHA, PUSHAD</b> - Push All to GPR Words or Doublewords 60			INVALID IN 64-BIT MODE (invalid-opcode exception)	
<b>PUSHF, PUSHFD, PUSHFQ</b> —Push rFLAGS Word, Doubleword, or Quadword onto Stack 9C	Promoted to 64 bits.	64 bits	Cannot encode <sup>6</sup>	PUSHFQ (new mnemonic): Pushes the 64-bit RFLAGS register.
<b>RCL</b> —Rotate Through Carry Left D1 /2 D3 /2 C1 /2	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
<b>RCR</b> —Rotate Through Carry Right D1 /3 D3 /3 C1 /3	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
<b>RDMSR</b> —Read Model-Specific Register 0F 32	Same as legacy mode.	Not relevant.	RDX[31:0] contains MSR[63:32], RAX[31:0] contains MSR[31:0]. Zero-extends 32-bit register results to 64 bits.	
<b>Notes:</b>				
<ol style="list-style-type: none"> <li>1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.</li> <li>2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.</li> <li>3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</li> <li>4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.</li> <li>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</li> <li>6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</li> </ol>				

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>RDPMC</b> —Read Performance-Monitoring Counters 0F 33	Same as legacy mode.	Not relevant.	RDX[31:0] contains PMC[63:32], RAX[31:0] contains PMC[31:0]. Zero-extends 32-bit register results to 64 bits.	
<b>RDTSC</b> —Read Time-Stamp Counter 0F 31	Same as legacy mode.	Not relevant.	RDX[31:0] contains TSC[63:32], RAX[31:0] contains TSC[31:0]. Zero-extends 32-bit register results to 64 bits.	
<b>RDTSCP</b> —Read Time-Stamp Counter and Processor ID 0F 01 F9	Same as legacy mode.	Not relevant.	RDX[31:0] contains TSC[63:32], RAX[31:0] contains TSC[31:0]. RCX[31:0] contains the TSC_AUX MSR C000_0103h[31:0]. Zero-extends 32-bit register results to 64 bits.	
<b>REP INS</b> —Repeat Input String F3 6D	Same as legacy mode.	32 bits	Reads doubleword I/O port. See footnote <sup>5</sup>	
<b>REP LODS</b> —Repeat Load String F3 AD	Promoted to 64 bits.	32 bits	Zero-extends EAX to 64 bits. See footnote <sup>5</sup>	See footnote <sup>5</sup>
<b>REP MOVS</b> —Repeat Move String F3 A5	Promoted to 64 bits.	32 bits	No GPR register results. See footnote <sup>5</sup>	
<b>REP OUTS</b> —Repeat Output String to Port F3 6F	Same as legacy mode.	32 bits	Writes doubleword to I/O port. No GPR register results. See footnote <sup>5</sup>	
<b>REP STOS</b> —Repeat Store String F3 AB	Promoted to 64 bits.	32 bits	No GPR register results. See footnote <sup>5</sup>	
<b>REPx CMPS</b> —Repeat Compare String F3 A7	Promoted to 64 bits.	32 bits	No GPR register results. See footnote <sup>5</sup>	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>REPx SCAS</b> —Repeat Scan String F3 AF	Promoted to 64 bits.	32 bits	No GPR register results. See footnote <sup>5</sup>	
<b>RET</b> —Return from Call Near C2 C3	See “Near Branches in 64-Bit Mode” in Volume 1.			
	Promoted to 64 bits.	64 bits	Cannot encode. <sup>6</sup>	No GPR register results.
<b>RET</b> —Return from Call Far CB CA	Promoted to 64 bits.	32 bits	See “Control Transfers” in Volume 1 and “Control-Transfer Privilege Checks” in Volume 2.	
<b>ROL</b> —Rotate Left D1 /0 D3 /0 C1 /0	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	Uses 6-bit count.
<b>ROR</b> —Rotate Right D1 /1 D3 /1 C1 /1	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	Uses 6-bit count.
<b>RSM</b> —Resume from System Management Mode 0F AA	New SMM state-save area.	Not relevant.	See “System-Management Mode” in Volume 2.	
<b>SAHF</b> —Store AH into Flags 9E	Same as leg- acy mode.	Not relevant.	No GPR register results.	
<b>SAL</b> —Shift Arithmetic Left D1 /4 D3 /4 C1 /4	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	Uses 6-bit count.

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>SAR</b> —Shift Arithmetic Right D1 /7 D3 /7 C1 /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
<b>SBB</b> —Subtract with Borrow 19 1B 1D 81 /3 83 /3	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
<b>SCAS, SCASW, SCASD, SCASQ</b> — Scan String AF	Promoted to 64 bits.	32 bits	SCASD: Scan String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote <sup>5</sup>	SCASQ (new mnemonic): Scan String Quadwords. See footnote <sup>5</sup>
<b>SFENCE</b> —Store Fence 0F AE /7	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>SGDT</b> —Store Global Descriptor Table Register 0F 01 /0	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Stores 8-byte base and 2-byte limit.	
<b>SHL</b> —Shift Left D1 /4 D3 /4 C1 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>SHLD</b> —Shift Left Double 0F A4 0F A5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
<b>SHR</b> —Shift Right D1 /5 D3 /5 C1 /5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
<b>SHRD</b> —Shift Right Double 0F AC 0F AD	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
<b>SIDT</b> —Store Interrupt Descriptor Table Register 0F 01 /1	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Stores 8-byte base and 2-byte limit.	
<b>SKINIT</b> —Secure Init and Jump with Attestation 0F 01 DE	Same as legacy mode.	Not relevant	Zero-extends 32-bit register results to 64 bits.	
<b>SLDT</b> —Store Local Descriptor Table Register 0F 00 /0	Same as legacy mode.	32	Zero-extends 2-byte LDT selector to 64 bits.	
<b>SMSW</b> —Store Machine Status Word 0F 01 /4	Same as legacy mode.	32	Zero-extends 32-bit register results to 64 bits.	Stores 64-bit machine status word (CR0).
<b>STC</b> —Set Carry Flag F9	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>STD</b> —Set Direction Flag FD	Same as legacy mode.	Not relevant.	No GPR register results.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>STGI</b> —Set Global Interrupt Flag 0F 01 DC	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>STI</b> - Set Interrupt Flag FB	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>STOS, STOSW, STOSD, STOSQ</b> - Store String  AB	Promoted to 64 bits.	32 bits	STOSD: Store String Doublewords. See footnote <sup>5</sup>	STOSQ (new mnemonic): Store String Quadwords. See footnote <sup>5</sup>
<b>STR</b> —Store Task Register 0F 00 /1	Same as legacy mode.	32	Zero-extends 2-byte TR selector to 64 bits.	
<b>SUB</b> —Subtract 29 2B 2D 81 /5 83 /5	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	
<b>SWAPGS</b> —Swap GS Register with KernelGSbase MSR  0F 01 /7	New instruction, available only in 64-bit mode. (In other modes, this opcode is invalid.)	Not relevant.	See “SWAPGS Instruction” in Volume 2.	
<b>SYSCALL</b> —Fast System Call 0F 05	Promoted to 64 bits.	Not relevant.	See “SYSCALL and SYSRET Instructions” in Volume 2 for details.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>SYSENTER</b> —System Call 0F 34	INVALID IN LONG MODE (invalid-opcode exception)			
<b>SYSEXIT</b> —System Return 0F 35	INVALID IN LONG MODE (invalid-opcode exception)			
<b>SYSRET</b> —Fast System Return 0F 07	Promoted to 64 bits.	32 bits	See “SYSCALL and SYSRET Instructions” in Volume 2 for details.	
<b>TEST</b> —Test Bits 85 A9 F7 /0	Promoted to 64 bits.	32 bits	No GPR register results.	
<b>UD2</b> —Undefined Operation 0F 0B	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>VERR</b> —Verify Segment for Reads 0F 00 /4	Same as legacy mode.	Operand size fixed at 16 bits	No GPR register results.	
<b>VERW</b> —Verify Segment for Writes 0F 00 /5	Same as legacy mode.	Operand size fixed at 16 bits	No GPR register results.	
<b>VMLOAD</b> —Load State from VMCB 0F 01 DA	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>VMMCALL</b> —Call VMM 0F 01 D9	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>VMRUN</b> —Run Virtual Machine 0F 01 D8	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>VMSAVE</b> —Save State to VMCB 0F 01 DB	Same as legacy mode.	Not relevant.	No GPR register results.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

**Table B-1. Operations and Operands in 64-Bit Mode (continued)**

Instruction and Opcode (hex) <sup>1</sup>	Type of Operation <sup>2</sup>	Default Operand Size <sup>3</sup>	For 32-Bit Operand Size <sup>4</sup>	For 64-Bit Operand Size <sup>4</sup>
<b>WAIT</b> —Wait for Interrupt 9B	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>WBINVD</b> —Writeback and Invalidate All Caches 0F 09	Same as legacy mode.	Not relevant.	No GPR register results.	
<b>WRMSR</b> —Write to Model-Specific Register 0F 30	Same as legacy mode.	Not relevant.	No GPR register results. MSR[63:32] = RDX[31:0] MSR[31:0] = RAX[31:0]	
<b>XADD</b> —Exchange and Add 0F C1	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	
<b>XCHG</b> —Exchange Register/Memory with Register 87 90	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	
<b>XOR</b> —Logical Exclusive OR 31 33 35 81 /6 83 /6	Promoted to 64 bits.	32 bits	Zero-extends 32- bit register results to 64 bits.	

**Notes:**

1. See “General Rules for 64-Bit Mode” on page 499, for opcodes that do not appear in this table.
2. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 499 for definitions of “Promoted to 64 bits” and related topics.
3. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
4. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediate and branch displacements are sign-extended to 64 bits.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.

## B.3 Invalid and Reassigned Instructions in 64-Bit Mode

Table B-2 lists instructions that are illegal in 64-bit mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

**Table B-2. Invalid Instructions in 64-Bit Mode**

Mnemonic	Opcode (hex)	Description
AAA	37	ASCII Adjust After Addition
AAD	D5	ASCII Adjust Before Division
AAM	D4	ASCII Adjust After Multiply
AAS	3F	ASCII Adjust After Subtraction
BOUND	62	Check Array Bounds
CALL (far)	9A	Procedure Call Far (far absolute)
DAA	27	Decimal Adjust after Addition
DAS	2F	Decimal Adjust after Subtraction
INTO	CE	Interrupt to Overflow Vector
JMP (far)	EA	Jump Far (absolute)
LDS	C5	Load DS Far Pointer
LES	C4	Load ES Far Pointer
POP DS	1F	Pop Stack into DS Segment
POP ES	07	Pop Stack into ES Segment
POP SS	17	Pop Stack into SS Segment
POPA, POPAD	61	Pop All to GPR Words or Doublewords
PUSH CS	0E	Push CS Segment Selector onto Stack
PUSH DS	1E	Push DS Segment Selector onto Stack
PUSH ES	06	Push ES Segment Selector onto Stack
PUSH SS	16	Push SS Segment Selector onto Stack
PUSHA, PUSHAD	60	Push All to GPR Words or Doublewords
Redundant Grp1	82 /2	Redundant encoding of group1 Eb,lb opcodes
SALC	D6	Set AL According to CF

Table B-3 lists instructions that are reassigned to different functions in 64-bit mode. Attempted use of these instructions generates the reassigned function.

**Table B-3. Reassigned Instructions in 64-Bit Mode**

Mnemonic	Opcode (hex)	Description
ARPL	63	Opcode for MOVSXD instruction in 64-bit mode. In all other modes, this is the Adjust Requestor Privilege Level instruction opcode.
DEC and INC	40-4F	REX prefixes in 64-bit mode. In all other modes, decrement by 1 and increment by 1.
LDS	C5	VEX Prefix. Introduces the VEX two-byte instruction encoding escape sequence.
LES	C4	VEX Prefix. Introduces the VEX three-byte instruction encoding escape sequence.

Table B-4 lists instructions that are illegal in long mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

**Table B-4. Invalid Instructions in Long Mode**

Mnemonic	Opcode (hex)	Description
SYSENTER	0F 34	System Call
SYSEXIT	0F 35	System Return

## B.4 Instructions with 64-Bit Default Operand Size

In 64-bit mode, two groups of instructions default to 64-bit operand size without the need for a REX prefix:

- *Near branches*—CALL, Jcc, JrCX, JMP, LOOP, and RET.
- *All instructions, except far branches, that implicitly reference the RSP*—CALL, ENTER, LEAVE, POP, PUSH, and RET (CALL and RET are in both groups of instructions).

Table B-5 lists these instructions.

**Table B-5. Instructions Defaulting to 64-Bit Operand Size**

Mnemonic	Opcode (hex)	Implicitly Reference RSP	Description
CALL	E8, FF /2	yes	Call Procedure Near
ENTER	C8	yes	Create Procedure Stack Frame
Jcc	many	no	Jump Conditional Near
JMP	E9, EB, FF /4	no	Jump Near
LEAVE	C9	yes	Delete Procedure Stack Frame
LOOP	E2	no	Loop
LOOPcc	E0, E1	no	Loop Conditional
POP reg/mem	8F /0	yes	Pop Stack (register or memory)
POP reg	58-5F	yes	Pop Stack (register)
POP FS	0F A1	yes	Pop Stack into FS Segment Register
POP GS	0F A9	yes	Pop Stack into GS Segment Register
POPF, POPFD, POPFQ	9D	yes	Pop to rFLAGS Word, Doubleword, or Quadword
PUSH imm8	6A	yes	Push onto Stack (sign-extended byte)
PUSH imm32	68	yes	Push onto Stack (sign-extended doubleword)
PUSH reg/mem	FF /6	yes	Push onto Stack (register or memory)
PUSH reg	50-57	yes	Push onto Stack (register)
PUSH FS	0F A0	yes	Push FS Segment Register onto Stack
PUSH GS	0F A8	yes	Push GS Segment Register onto Stack
PUSHF, PUSHFD, PUSHFQ	9C	yes	Push rFLAGS Word, Doubleword, or Quadword onto Stack
RET	C2, C3	yes	Return From Call (near)

The 64-bit default operand size can be overridden to 16 bits using the 66h operand-size override. However, it is not possible to override the operand size to 32 bits because there is no 32-bit operand-size override prefix for 64-bit mode. See “Operand-Size Override Prefix” on page 7 for details.

## B.5 Single-Byte INC and DEC Instructions in 64-Bit Mode

In 64-bit mode, the legacy encodings for the 16 single-byte INC and DEC instructions (one for each of the eight GPRs) are used to encode the REX prefix values, as described in “REX Prefix” on page 14. Therefore, these single-byte opcodes for INC and DEC are not available in 64-bit mode, although they are available in legacy and compatibility modes. The functionality of these INC and DEC instructions is still available in 64-bit mode, however, using the ModRM forms of those instructions (opcodes FF/0 and FF/1).

## B.6 NOP in 64-Bit Mode

Programs written for the legacy x86 architecture commonly use opcode 90h (the XCHG EAX, EAX instruction) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h specially in order to preserve this legacy NOP use. Without special handling in 64-bit mode, the instruction would not be a true no-operation. Therefore, in 64-bit mode the processor treats XCHG EAX, EAX as a true NOP, regardless of operand size.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction. Unless a 64-bit operand size is specified using a REX prefix byte, using the two byte form of XCHG to exchange a register with itself will not result in a no-operation because the default operation size is 32 bits in 64-bit mode.

## B.7 Segment Override Prefixes in 64-Bit Mode

In 64-bit mode, the CS, DS, ES, SS segment-override prefixes have no effect. These four prefixes are no longer treated as segment-override prefixes in the context of multiple-prefix rules. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as true segment-override prefixes in 64-bit mode. Use of the FS and GS prefixes cause their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” in Volume 2 for details.

## Appendix C Differences Between Long Mode and Legacy Mode

Table C-1 summarizes the major differences between 64-bit mode and legacy protected mode. The third column indicates differences between 64-bit mode and legacy mode. The fourth column indicates whether that difference also applies to compatibility mode.

**Table C-1. Differences Between Long Mode and Legacy Mode**

Type	Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
Application Programming	Addressing	RIP-relative addressing available	no
		Default data size is 32 bits	
	Data and Address Sizes	REX Prefix toggles data size to 64 bits	
		Default address size is 64 bits	
		Address size prefix toggles address size to 32 bits	
	Instruction Differences	Various opcodes are invalid or changed in 64-bit mode (see Table B-2 on page 525 and Table B-3 on page 526)	
		Various opcodes are invalid in long mode (see Table B-4 on page 526)	yes
		MOV reg,imm32 becomes MOV reg,imm64 (with REX operand size prefix)	no
		REX is always enabled	
		Direct-offset forms of MOV to or from accumulator become 64-bit offsets	
		MOVD extended to MOV 64 bits between MMX registers and long GPRs (with REX operand-size prefix)	

**Table C-1. Differences Between Long Mode and Legacy Mode (continued)**

Type	Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
System Programming	x86 Modes	Real and virtual-8086 modes not supported	yes
	Task Switching	Task switching not supported	yes
	Addressing	64-bit virtual addresses	yes
		4-level paging structures	
		PAE must always be enabled	
	Segmentation	CS, DS, ES, SS segment bases are ignored	no
		CS, DS, ES, FS, GS, SS segment limits are ignored	
		CS, DS, ES, SS Segment prefixes are ignored	
	Exception and Interrupt Handling	All pushes are 8 bytes	yes
		16-bit interrupt and trap gates are illegal	
		32-bit interrupt and trap gates are redefined as 64-bit gates and are expanded to 16 bytes	
		SS is set to null on stack switch	
		SS:RSP is pushed unconditionally	
	Call Gates	All pushes are 8 bytes	yes
		16-bit call gates are illegal	
		32-bit call gate type is redefined as 64-bit call gate and is expanded to 16 bytes.	
		SS is set to null on stack switch	
	System-Descriptor Registers	GDT, IDT, LDT, TR base registers expanded to 64 bits	yes
	System-Descriptor Table Entries and Pseudo-descriptors	LGDT and LIDT use expanded 10-byte pseudo-descriptors.	no
		LLDT and LTR use expanded 16-byte table entries.	

## Appendix D Instruction Subsets and CPUID Feature Flags

This appendix provides information that can be used to determine if a specific instruction within the AMD64 instruction-set architecture (ISA) is supported on a processor.

Originally the x86 ISA was composed of a set of instructions from the general-purpose and system instruction groups. This set forms the base of the AMD64 ISA. As the ISA expanded over time, new instructions were added. Each addition constituted either a single instruction or a set of instructions and each addition was assigned a specific processor feature flag.

Although most current processor products support the entire ISA, support for each added instruction or instruction subset is optional and must be confirmed by testing the corresponding feature flag. The presence of a particular instruction or subset is indicated by the corresponding feature flag being set. A feature flag is a single bit value located at a specific bit position within the 32-bit value returned in a register as a result of executing the CPUID instruction.

For more information on using the CPUID instruction, see the instruction reference page for CPUID on page 158. For a comprehensive list of processor feature flags accessed using the CPUID instruction, see Appendix E, “Obtaining Processor Information Via the CPUID Instruction” on page 601.

## D.1 Instruction Set Overview

The AMD64 ISA can be organized into five instruction groups:

### 1. General-purpose instructions

These instructions operate on the general-purpose registers (GP registers) and can be used at all privilege levels. This group includes instructions to load and store the contents of a GP register to and from memory, move values between the GP registers, and perform arithmetic and logical operations on the contents of the registers.

### 2. System instructions

These instructions provide the means to manipulate the processor operating mode, access processor resources, handle program and system errors, and manage system memory. Many of these instructions require privilege level 0 to execute.

### 3. x87 instructions

These instructions are available at all privilege levels and include legacy floating-point instructions that use the ST(0)–ST(7) stack registers (FPR0–FPR7 physical registers) and internally use extended precision (80-bit) binary floating-point representation and operations.

### 4. 64-bit media Instructions

These instructions are available at all privilege levels and perform vector operations on packed integer and floating-point values held in the 64-bit MMX™ registers. The MMX register set overlays the FPR0–FPR7 physical registers. This group is composed of the MMX and 3DNow!™ instruction subsets and was subsequently expanded by the MMX and 3DNow! extensions subsets.

### 5. SSE instructions

The SSE instructions operate on packed integer and floating-point values held in the XMM / YMM registers. SSE includes the original Streaming SIMD Extensions, all the subsequent named SSE subsets, and the AVX, XOP, and AES instructions.

Figure D-1 on page 533 represents the relationship between the five major instruction groups and the named instruction subsets. Circles represent the instruction subsets. These include the base instruction set labeled “Base Instructions” in the diagram and the named subsets. The diagram omits individual optional instructions and some of the minor named instruction subsets. Dashed-line polygons represent the instruction groups.

Note that the 128-bit and 256-bit media instructions are referred to collectively as the Streaming SIMD Extensions (SSE). This is also the name of the original SSE subset. In the diagram the original SSE subset is labeled “SSE1 Instructions.” Collectively the 64-bit media and the SSE instructions make up the single instruction / multiple data (SIMD) group (labeled “SIMD Instructions” in the diagram).

The overlapping of the SSE and 64-bit media instruction subsets indicates that these subsets share some common mnemonics. However, these common mnemonics either have distinct opcodes for each subset or they take operands in both the MMX and XMM register sets.

The horizontal axis of Figure D-1 shows how the subsets have evolved over time.

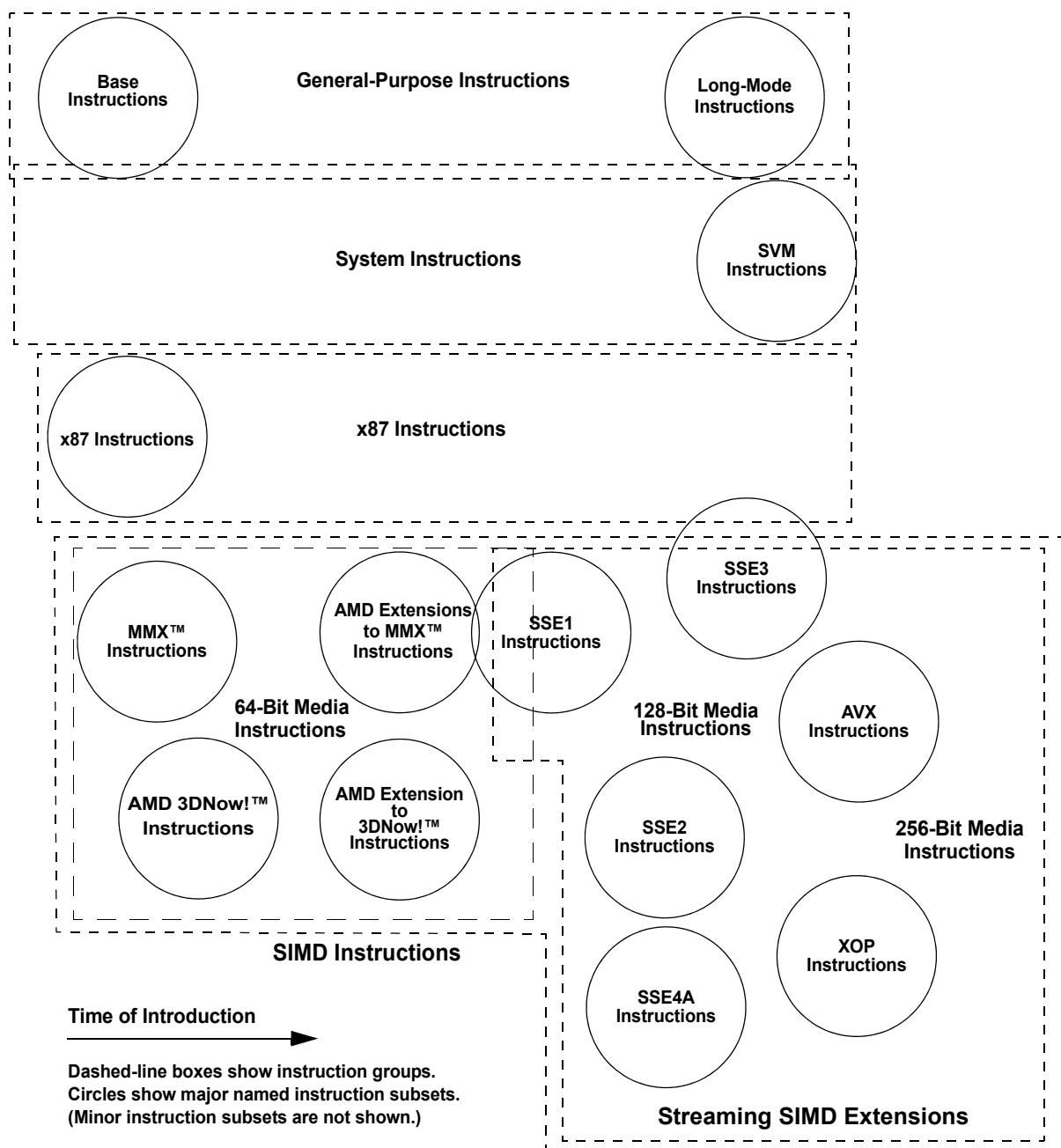


Figure D-1. AMD64 ISA Instruction Subsets

## D.2 CPUID Feature Flags Related to Instruction Support

Only a subset of the CPUID feature flags provides information related to instruction support.

The feature flags related to supported instruction subsets are accessed via the *standard* function number 0000\_0001h, the *extended* function number 8000\_00001h, and the *structured extended* function number 0000\_0007h.

The following table lists all flags related to instruction support. Entries for each flag provide the instruction or instruction subset corresponding to the flag, the CPUID function that must be executed to access the flag, and the bit position of the flag in the return value. The feature flags listed are used in Table D-2 on page 537:

**Table D-1. Feature Flags for Instruction / Instruction Subset Support**

Feature Flag	Instruction or Subset	CPUID Function <sup>1</sup>	Feature Flag Bit Position <sup>2</sup>
BASE	Base Instruction set	—	—
CLFSH	CLFLUSH	standard	EDX[19]
CMPXCHG8B	CMPXCHG8B	both	EDX[8]
CMPXCHG16B	CMPXCHG16B	standard	ECX[13]
CMOV	CMOVcc	both	EDX[15]
MSR	RDMSR / WRMSR	both	EDX[5]
TSC	RDTSC / RDTSCP	both	EDX[4]
RDTSCP	RDTSCP	extended	EDX[27]
SysCallSysRet	SYSCALL / SYSRET	extended	EDX[11]
SysEnterSysExit	SYSENTER / SYSEXIT	standard	EDX[11]
FPU	x87	both	EDX[0]
x87 && CMOV	FCMOVcc <sup>3</sup>	both	EDX[0] && EDX[15]
MMX	MMX	both	EDX[23]
3DNow	3DNow!	extended	EDX[31]
MmxExt	MMX Extensions	extended	EDX[22]
3DNowExt	3DNow! Extensions	extended	EDX[30]
3DNowPrefetch	PREFETCH / PREFETCHW	extended	ECX[8]
SSE	SSE1	standard	EDX[25]
SSE2	SSE2	standard	EDX[26]

**Notes:**

1. *standard* = Fn 0000\_0001h; *extended* = Fn 8000\_00001h; *both* means that both standard and extended CPUID functions return the same feature flag in the same bit position of the return value. For functions of the form xxxx\_xxxx\_x, the trailing digit is the value required in ECX.
2. Register and bit position of the return value that corresponds to the feature flag.
3. FCMOVcc instruction is supported if x87 and CMOVcc instructions are both supported.
4. XSAVE (and related) instructions require separate enablement.

**Table D-1. Feature Flags for Instruction / Instruction Subset Support**

Feature Flag	Instruction or Subset	CPUID Function <sup>1</sup>	Feature Flag Bit Position <sup>2</sup>
SSE3	SSE3	standard	ECX[0]
SSSE3	SSSE3	standard	ECX[9]
SSE4A	SSE4A	extended	ECX[6]
SSE41	SSE4.1	standard	ECX[19]
SSE42	SSE4.2	standard	ECX[20]
LM	Long Mode	extended	EDX[29]
SVM	Secure Virtual Machine	extended	ECX[2]
AVX	AVX	standard	ECX[28]
AVX2	AVX2	0000_0007_0	EBX[5]
XOP	XOP	extended	ECX[11]
AES	AES	standard	ECX[25]
FMA	FMA	standard	ECX[12]
FMA4	FMA4	extended	ECX[16]
F16C	16-bit floating-point conversion	standard	ECX[29]
RDRAND	RDRAND	standard	ECX[30]
ABM	LZCNT	extended	ECX[5]
BMI1	Bit Manipulation, group 1	0000_0007_0	EBX[3]
BMI2	Bit Manipulation, group 2	0000_0007_0	EBX[8]
POPCNT	POPCNT	standard	ECX[23]
TBM	Trailing bit manipulation	extended	ECX[21]
MOVBE	MOVBE	standard	ECX[22]
MONITOR	MONITOR / MWAIT	standard	ECX[3]
MONITORX	MONITORX / MWAITX	extended	ECX[29]
PCLMULQDQ	PCLMULQDQ	standard	ECX[1]
FXSR	FXSAVE / FXRSTOR	both	EDX[24]
SKINIT	SKINIT / STGI	extended	ECX[12]
LahfSahf	LAHF / SAHF	extended	ECX[0]
FSGSBASE	FS and GS base read and write	0000_0007_0	EBX[0]
SHA	SHA	0000_0007_0	EBX[29]
CLFLOPT	CLFLOPT	0000_0007_0	EBX[23]
SMAP	SMAP	0000_0007_0	EBX[20]
ADX	ADX	0000_0007_0	EBX[19]

**Notes:**

1. standard = Fn 0000\_0001h; extended = Fn 8000\_0001h; both means that both standard and extended CPUID functions return the same feature flag in the same bit position of the return value. For functions of the form xxxx\_xxxx\_x, the trailing digit is the value required in ECX.
2. Register and bit position of the return value that corresponds to the feature flag.
3. FCMOVcc instruction is supported if x87 and CMOVcc instructions are both supported.
4. XSAVE (and related) instructions require separate enablement.

**Table D-1. Feature Flags for Instruction / Instruction Subset Support**

Feature Flag	Instruction or Subset	CPUID Function <sup>1</sup>	Feature Flag Bit Position <sup>2</sup>
RDSEED	RDSEED	0000_0007_0	EBX[18]
SME	SME	8000_001F	EAX[0]
SEV	SEV	8000_001F	EAX[1]
PageFlushMsr	PageFlushMsr	8000_001F	EAX[2]
ES	ES	8000_001F	EAX[3]
CLZERO	CLZERO	8000_0008	EBX[0]
Instruction Retired Counter	Instruction Retired Counter	8000_0008	EBX[1]
Error Pointer Zero/Restore	Error Pointer Zero/Restore	8000_0008	EBX[2]
XSAVEOPT	XSAVEOPT	0000_000D_1	EAX[0]
XSAVEC	XSAVEC	0000_000D_1	EAX[1]
XGETBV w/ ECX=1	XGETBV w/ ECX=1	0000_000D_1	EAX[2]
XSAVES/XRSTORS	XSAVES/XRSTORS	0000_000D_1	EAX[3]
XSAVE	XSAVE / XRSTOR <sup>4</sup>	standard	ECX[26]

**Notes:**

1. *standard* = Fn 0000\_0001h; *extended* = Fn 8000\_0001h; *both* means that both standard and extended CPUID functions return the same feature flag in the same bit position of the return value. For functions of the form xxxx\_xxxx\_x, the trailing digit is the value required in ECX.
2. Register and bit position of the return value that corresponds to the feature flag.
3. FCMOVcc instruction is supported if x87 and CMOVcc instructions are both supported.
4. XSAVE (and related) instructions require separate enablement.

## D.3 Instruction List

Table D-2 shows the minimum current privilege level (CPL) required to execute each instruction and the feature flag or flags that indicates support for that instruction. Each flag is listed in the column corresponding to the instruction group to which it belongs. Note that some instructions span groups.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
AAA	ASCII Adjust After Addition	3	Base				
AAD	ASCII Adjust Before Division	3	Base				
AAM	ASCII Adjust After Multiply	3	Base				
AAS	ASCII Adjust After Subtraction	3	Base				
ADC	Add with Carry	3	Base				
ADD	Signed or Unsigned Add	3	Base				
ADDPD	Add Packed Double-Precision Floating-Point	3		SSE2			
ADDPS	Add Packed Single-Precision Floating-Point	3		SSE			
ADDSD	Add Scalar Double-Precision Floating-Point	3		SSE2			
ADDSS	Add Scalar Single-Precision Floating-Point	3		SSE			
ADDSUBPD	Add and Subtract Double-Precision	3		SSE3			
ADDSUBPS	Add and Subtract Single-Precision	3		SSE3			
AESDEC	AES Decryption Round	3		AES			
AESDECLAST	AES Last Decryption Round	3		AES			
AESENC	AES Encryption Round	3		AES			
AESENCLAST	AES Last Encryption Round	3		AES			
AESIMC	AES InvMixColumn Transformation	3		AES			
AESKEYGENASSIST	AES Assist Round Key Generation	3		AES			
AND	Logical AND	3	Base				
ANDN	Logical And-Not	3	BMI1				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
ANDNPD	AND NOT Packed Double-Precision Floating-Point	3		SSE2			
ANDNPS	AND NOT Packed Single-Precision Floating-Point	3		SSE			
ANDPD	AND Packed Double-Precision Floating-Point	3		SSE2			
ANDPS	AND Packed Single-Precision Floating-Point	3		SSE			
ARPL	Adjust Requestor Privilege Level	3					Base
BEXTR (immediate form)	Bit Field Extract	3	TBM				
BEXTR (register form)	Bit Field Extract	3	BMI1				
BLCFILL	Fill From Lowest Clear Bit	3	TBM				
BLCI	Isolate Lowest Clear Bit	3	TBM				
BLSI	Isolate Lowest Set Bit	3	BMI1				
BLCIC	Isolate Lowest Clear Bit and Complement	3	TBM				
BLCMSK	Mask From Lowest Clear Bit	3	TBM				
BLCS	Set Lowest Clear Bit	3	TBM				
BLENDPD	Blend Packed Double-Precision Floating-Point	3		SSE41			
BLENDPS	Blend Packed Single-Precision Floating-Point	3		SSE41			
BLENDVPD	Variable Blend Packed Double-Precision Floating-Point	3		SSE41			
BLENDVPS	Variable Blend Packed Single-Precision Floating-Point	3		SSE41			
BLSMSK	Mask From Lowest Set Bit	3	BMI1				
BLSFILL	Fill From Lowest Set Bit	3	TBM				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
BLSIC	Isolate Lowest Set Bit and Complement	3	TBM				
BLSR	Reset Lowest Set Bit	3	BMI1				
BOUND	Check Array Bounds	3	Base				
BSF	Bit Scan Forward	3	Base				
BSR	Bit Scan Reverse	3	Base				
BSWAP	Byte Swap	3	Base				
BT	Bit Test	3	Base				
BTC	Bit Test and Complement	3	Base				
BTR	Bit Test and Reset	3	Base				
BTS	Bit Test and Set	3	Base				
BZHI	Zero High Bits	3	BMI2				
CALL	Procedure Call	3	Base				
CBW	Convert Byte to Word	3	Base				
CDQ	Convert Doubleword to Quadword	3	Base				
CDQE	Convert Doubleword to Quadword	3	LM				
CLC	Clear Carry Flag	3	Base				
CLD	Clear Direction Flag	3	Base				
CLFLUSH	Cache Line Flush	3	CLFSH				
CLGI	Clear Global Interrupt Flag	0					SVM
CLI	Clear Interrupt Flag	3					Base
CLTS	Clear Task-Switched Flag in CR0	0					Base
CMC	Complement Carry Flag	3	Base				
CMOVcc	Conditional Move	3	CMOV				
CMP	Compare	3	Base				
CMPPD	Compare Packed Double-Precision Floating-Point	3		SSE2			
CMPPS	Compare Packed Single-Precision Floating-Point	3		SSE			
CMPS	Compare Strings	3	Base				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>					
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System	
CMPSB	Compare Strings by Byte	3	Base					
CMPSD	Compare Strings by Doubleword	3	Base <sup>2</sup>					
CMPSD	Compare Scalar Double-Precision Floating-Point	3		SSE2 <sup>2</sup>				
CMPSQ	Compare Strings by Quadword	3	LM					
CMPSS	Compare Scalar Single-Precision Floating-Point	3		SSE				
CMPSW	Compare Strings by Word	3	Base					
CMPXCHG	Compare and Exchange	3	Base					
CMPXCHG8B	Compare and Exchange Eight Bytes	3	CMPXCHG8B					
CMPXCHG16B	Compare and Exchange Sixteen Bytes	3	CMPXCHG16B					
COMISD	Compare Ordered Scalar Double-Precision Floating-Point	3		SSE2				
COMISS	Compare Ordered Scalar Single-Precision Floating-Point	3		SSE				
CPUID	Processor Identification	3	Base					
CQO	Convert Quadword to Double Quadword	3	LM					
CRC32	32-bit Cyclical Redundancy Check	3	SSE42					
CVTDQ2PD	Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point	3		SSE2				
CVTDQ2PS	Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point	3		SSE2				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
CVTPD2DQ	Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers	3		SSE2			
CVTPD2PI	Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers	3		SSE2	SSE2		
CVTPD2PS	Convert Packed Double-Precision Floating-Point to Packed Single-Precision Floating-Point	3		SSE2			
CVTPI2PD	Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point	3		SSE2	SSE2		
CVTPI2PS	Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point	3		SSE	SSE		
CVTPS2DQ	Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers	3		SSE2			
CVTPS2PD	Convert Packed Single-Precision Floating-Point to Packed Double-Precision Floating-Point	3		SSE2			
CVTPS2PI	Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers	3		SSE	SSE		
CVTSD2SI	Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer	3		SSE2			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
CVTSD2SS	Convert Scalar Double-Precision Floating-Point to Scalar Single-Precision Floating-Point	3		SSE2			
CVTSI2SD	Convert Signed Doubleword or Quadword Integer to Scalar Double-Precision Floating-Point	3		SSE2			
CVTSI2SS	Convert Signed Doubleword or Quadword Integer to Scalar Single-Precision Floating-Point	3		SSE			
CVTSS2SD	Convert Scalar Single-Precision Floating-Point to Scalar Double-Precision Floating-Point	3		SSE2			
CVTSS2SI	Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer	3		SSE			
CVTPD2DQ	Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated	3		SSE2			
CVTPD2PI	Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated	3		SSE2	SSE2		
CVTPS2DQ	Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated	3		SSE2			
CVTPS2PI	Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated	3		SSE	SSE		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
CVTTSD2SI	Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated	3		SSE2			
CVTTSS2SI	Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated	3		SSE			
CWD	Convert Word to Doubleword	3	Base				
CWDE	Convert Word to Doubleword	3	Base				
DAA	Decimal Adjust after Addition	3	Base				
DAS	Decimal Adjust after Subtraction	3	Base				
DEC	Decrement by 1	3	Base				
DIV	Unsigned Divide	3	Base				
DIVPD	Divide Packed Double-Precision Floating-Point	3		SSE2			
DIVPS	Divide Packed Single-Precision Floating-Point	3		SSE			
DIVSD	Divide Scalar Double-Precision Floating-Point	3		SSE2			
DIVSS	Divide Scalar Single-Precision Floating-Point	3		SSE			
DPPD	Dot Product Packed Double-Precision Floating-Point	3		SSE41			
DPPS	Dot Product Packed Single-Precision Floating-Point	3		SSE41			
EMMS	Enter/Exit Multimedia State	3			MMX	MMX	

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
ENTER	Create Procedure Stack Frame	3	Base				
EXTRACTPS	Extract Packed Single-Precision Floating-Point	3		SSE41			
EXTRQ	Extract Field From Register	3		SSE4A			
F2XM1	Floating-Point Compute 2x–1	3				X87	
FABS	Floating-Point Absolute Value	3				X87	
FADD	Floating-Point Add	3				X87	
FADDP	Floating-Point Add and Pop	3				X87	
FBLD	Floating-Point Load Binary-Coded Decimal	3				X87	
FBSTP	Floating-Point Store Binary-Coded Decimal Integer and Pop	3				X87	
FCHS	Floating-Point Change Sign	3				X87	
FCLEX	Floating-Point Clear Flags	3				X87	
FCMOVB	Floating-Point Conditional Move If Below	3				X87 && CMOV	
FCMOVBE	Floating-Point Conditional Move If Below or Equal	3				X87 && CMOV	
FCMOVE	Floating-Point Conditional Move If Equal	3				X87 && CMOV	
FCMOVNBB	Floating-Point Conditional Move If Not Below	3				X87 && CMOV	
FCMOVNBE	Floating-Point Conditional Move If Not Below or Equal	3				X87 && CMOV	
FCMOVNE	Floating-Point Conditional Move If Not Equal	3				X87 && CMOV	
FCMOVNU	Floating-Point Conditional Move If Not Unordered	3				X87 && CMOV	

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
FCMOVU	Floating-Point Conditional Move If Unordered	3				X87 && CMOV	
FCOM	Floating-Point Compare	3				X87	
FCOMI	Floating-Point Compare and Set Flags	3				X87	
FCOMIP	Floating-Point Compare and Set Flags and Pop	3				X87	
FCOMP	Floating-Point Compare and Pop	3				X87	
FCOMPP	Floating-Point Compare and Pop Twice	3				X87	
FCOS	Floating-Point Cosine	3				X87	
FDECSTP	Floating-Point Decrement Stack-Top Pointer	3				X87	
FDIV	Floating-Point Divide	3				X87	
FDIVP	Floating-Point Divide and Pop	3				X87	
FDIVR	Floating-Point Divide Reverse	3				X87	
FDIVRP	Floating-Point Divide Reverse and Pop	3				X87	
FEMMS	Fast Enter/Exit Multimedia State	3			3DNow	3DNow	
FFREE	Free Floating-Point Register	3				X87	
FIADD	Floating-Point Add Integer to Stack Top	3				X87	
FICOM	Floating-Point Integer Compare	3				X87	
FICOMP	Floating-Point Integer Compare and Pop	3				X87	
FIDIV	Floating-Point Integer Divide	3				X87	
FIDIVR	Floating-Point Integer Divide Reverse	3				X87	

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
FILD	Floating-Point Load Integer	3				X87	
FIMUL	Floating-Point Integer Multiply	3				X87	
FINCSTP	Floating-Point Increment Stack-Top Pointer	3				X87	
FINIT	Floating-Point Initialize	3				X87	
FIST	Floating-Point Integer Store	3				X87	
FISTP	Floating-Point Integer Store and Pop	3				X87	
FISTTP	Floating-Point Integer Truncate and Store	3				SSE3	
FISUB	Floating-Point Integer Subtract	3				X87	
FISUBR	Floating-Point Integer Subtract Reverse	3				X87	
FLD	Floating-Point Load	3				X87	
FLD1	Floating-Point Load +1.0	3				X87	
FLDCW	Floating-Point Load x87 Control Word	3				X87	
FLDENV	Floating-Point Load x87 Environment	3				X87	
FLDL2E	Floating-Point Load Log <sub>2</sub> e	3				X87	
FLDL2T	Floating-Point Load Log <sub>2</sub> 10	3				X87	
FLDLG2	Floating-Point Load Log <sub>10</sub> 2	3				X87	
FLDLN2	Floating-Point Load Ln 2	3				X87	
FLDPI	Floating-Point Load Pi	3				X87	
FLDZ	Floating-Point Load +0.0	3				X87	
FMUL	Floating-Point Multiply	3				X87	
FMULP	Floating-Point Multiply and Pop	3				X87	

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
FNCLEX	Floating-Point No-Wait Clear Flags	3				X87	
FNINIT	Floating-Point No-Wait Initialize	3				X87	
FNOP	Floating-Point No Operation	3				X87	
FNSAVE	Save No-Wait x87 and MMX State	3			X87	X87	
FNSTCW	Floating-Point No-Wait Store x87 Control Word	3				X87	
FNSTENV	Floating-Point No-Wait Store x87 Environment	3				X87	
FNSTSW	Floating-Point No-Wait Store x87 Status Word	3				X87	
FPATAN	Floating-Point Partial Arctangent	3				X87	
FPREM	Floating-Point Partial Remainder	3				X87	
FPREM1	Floating-Point Partial Remainder	3				X87	
FPTAN	Floating-Point Partial Tangent	3				X87	
FRNDINT	Floating-Point Round to Integer	3				X87	
FRSTOR	Restore x87 and MMX State	3			X87	X87	
FSAVE	Save x87 and MMX State	3			X87	X87	
FSCALE	Floating-Point Scale	3				X87	
FSIN	Floating-Point Sine	3				X87	
FSINCOS	Floating-Point Sine and Cosine	3				X87	
FSQRT	Floating-Point Square Root	3				X87	
FST	Floating-Point Store Stack Top	3				X87	

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
FSTCW	Floating-Point Store x87 Control Word	3				X87	
FSTENV	Floating-Point Store x87 Environment	3				X87	
FSTP	Floating-Point Store Stack Top and Pop	3				X87	
FSTSW	Floating-Point Store x87 Status Word	3				X87	
FSUB	Floating-Point Subtract	3				X87	
FSUBP	Floating-Point Subtract and Pop	3				X87	
FSUBR	Floating-Point Subtract Reverse	3				X87	
FSUBRP	Floating-Point Subtract Reverse and Pop	3				X87	
FTST	Floating-Point Test with Zero	3				X87	
FUCOM	Floating-Point Unordered Compare	3				X87	
FUCOMI	Floating-Point Unordered Compare and Set Flags	3				X87	
FUCOMIP	Floating-Point Unordered Compare and Set Flags and Pop	3				X87	
FUCOMP	Floating-Point Unordered Compare and Pop	3				X87	
FUCOMPP	Floating-Point Unordered Compare and Pop Twice	3				X87	
FWAIT	Wait for x87 Floating-Point Exceptions	3				X87	
FXAM	Floating-Point Examine	3				X87	
FXCH	Floating-Point Exchange	3				X87	
FXRSTOR	Restore XMM, MMX, and x87 State	3		FXSR	FXSR	FXSR	

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
FXSAVE	Save XMM, MMX, and x87 State	3		FXSR	FXSR	FXSR	
FXTRACT	Floating-Point Extract Exponent and Significand	3				X87	
FYL2X	Floating-Point $y * \log_2 x$	3				X87	
FYL2XP1	Floating-Point $y * \log_2(x + 1)$	3				X87	
HADDPD	Horizontal Add Packed Double	3		SSE3			
HADDPS	Horizontal Add Packed Single	3		SSE3			
HLT	Halt	0					Base
HSUBPD	Horizontal Subtract Packed Double	3		SSE3			
HSUBPS	Horizontal Subtract Packed Single	3		SSE3			
IDIV	Signed Divide	3	Base				
IMUL	Signed Multiply	3	Base				
IN	Input from Port	3	Base				
INC	Increment by 1	3	Base				
INS	Input String	3	Base				
INSB	Input String Byte	3	Base				
INSD	Input String Doubleword	3	Base				
INSERTPS	Insert Packed Single-Precision Floating-Point	3		SSE41			
INSERTQ	Insert Field	3		SSE4A			
INSW	Input String Word	3	Base				
INT	Interrupt to Vector	3	Base				
INT 3	Interrupt to Debug Vector	3					Base
INTO	Interrupt to Overflow Vector	3	Base				
INVD	Invalidate Caches	0					Base
INVLPG	Invalidate TLB Entry	0					Base

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
INVLPGA	Invalidate TLB Entry in a Specified ASID	0					SVM
IRET	Interrupt Return Word	3					Base
IRETD	Interrupt Return Doubleword	3					Base
IRETQ	Interrupt Return Quadword	3					LM
Jcc	Jump Condition	3	Base				
JCXZ	Jump if CX Zero	3	Base				
JECXZ	Jump if ECX Zero	3	Base				
JMP	Jump	3	Base				
JRCXZ	Jump if RCX Zero	3	Base				
LAHF	Load Status Flags into AH Register	3	LahfSahf				
LAR	Load Access Rights Byte	3					Base
LDDQU	Load Unaligned Double Quadword	3		SSE3			
LDMXCSR	Load MXCSR Control/Status Register	3		SSE			
LDS	Load DS Far Pointer	3	Base				
LEA	Load Effective Address	3	Base				
LEAVE	Delete Procedure Stack Frame	3	Base				
LES	Load ES Far Pointer	3	Base				
LFENCE	Load Fence	3	SSE2				
LFS	Load FS Far Pointer	3	Base				
LGDT	Load Global Descriptor Table Register	0					Base
LGS	Load GS Far Pointer	3	Base				
LIDT	Load Interrupt Descriptor Table Register	0					Base
LLDT	Load Local Descriptor Table Register	0					Base
LMSW	Load Machine Status Word	0					Base
LODS	Load String	3	Base				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
LODSB	Load String Byte	3	Base				
LODSD	Load String Doubleword	3	Base				
LODSQ	Load String Quadword	3	LM				
LODSW	Load String Word	3	Base				
LOOP	Loop	3	Base				
LOOPE	Loop if Equal	3	Base				
LOOPNE	Loop if Not Equal	3	Base				
LOOPNZ	Loop if Not Zero	3	Base				
LOOPZ	Loop if Zero	3	Base				
LSL	Load Segment Limit	3	Base				
LSS	Load SS Segment Register	3	Base				
LTR	Load Task Register	0					Base
LZCNT	Count Leading Zeros	3	ABM				
MASKMOVDQU	Masked Move Double Quadword Unaligned	3		SSE2			
MASKMOVQ	Masked Move Quadword	3			SSE    MmxExt		
MAXPD	Maximum Packed Double-Precision Floating-Point	3		SSE2			
MAXPS	Maximum Packed Single-Precision Floating-Point	3		SSE			
MAXSD	Maximum Scalar Double-Precision Floating-Point	3		SSE2			
MAXSS	Maximum Scalar Single-Precision Floating-Point	3		SSE			
MFENCE	Memory Fence	3	SSE2				
MINPD	Minimum Packed Double-Precision Floating-Point	3		SSE2			
MINPS	Minimum Packed Single-Precision Floating-Point	3		SSE			
MINSD	Minimum Scalar Double-Precision Floating-Point	3		SSE2			
MINSS	Minimum Scalar Single-Precision Floating-Point	3		SSE			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
MONITOR	Setup Monitor Address <sup>3</sup>	0					MONITOR
MONITORX	Setup Monitor Address	3	MONITORX				
MOV	Move	3	Base				
MOV CR $n$	Move to/from Control Registers	0					Base
MOV DR $n$	Move to/from Debug Registers	0					Base
MOVAPD	Move Aligned Packed Double-Precision Floating-Point	3		SSE2			
MOVAPS	Move Aligned Packed Single-Precision Floating-Point	3		SSE			
MOVBE	Move Big Endian	3	MOVBE				
MOVD	Move Doubleword or Quadword	3	MMX, SSE2	SSE2	MMX		
MOVDDUP	Move Double-Precision and Duplicate	3		SSE3			
MOVDQ2Q	Move Quadword to Quadword	3		SSE2	SSE2		
MOVDQA	Move Aligned Double Quadword	3		SSE2			
MOVDQU	Move Unaligned Double Quadword	3		SSE2			
MOVHLPS	Move Packed Single-Precision Floating-Point High to Low	3		SSE			
MOVHPD	Move High Packed Double-Precision Floating-Point	3		SSE2			
MOVHPS	Move High Packed Single-Precision Floating-Point	3		SSE			
MOVLHPS	Move Packed Single-Precision Floating-Point Low to High	3		SSE			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
MOVLPD	Move Low Packed Double-Precision Floating-Point	3		SSE2			
MOVLPS	Move Low Packed Single-Precision Floating-Point	3		SSE			
MOVMSKPD	Extract Packed Double-Precision Floating-Point Sign Mask	3	SSE2	SSE2			
MOVMSKPS	Extract Packed Single-Precision Floating-Point Sign Mask	3	SSE	SSE			
MOVNTDQ	Move Non-Temporal Double Quadword	3		SSE2			
MOVNDQA	Move Non-Temporal Double Quadword Aligned	3		SSE41			
MOVNTI	Move Non-Temporal Doubleword or Quadword	3	SSE2				
MOVNTPD	Move Non-Temporal Packed Double-Precision Floating-Point	3		SSE2			
MOVNTPS	Move Non-Temporal Packed Single-Precision Floating-Point	3		SSE			
MOVNTSD	Move Non-Temporal Scalar Double-Precision Floating-Point	3		SSE4A			
MOVNTSS	Move Non-Temporal Scalar Single-Precision Floating-Point	3		SSE4A			
MOVNTQ	Move Non-Temporal Quadword	3			SSE    MmxExt		
MOVQ	Move Quadword	3		SSE2	MMX		
MOVQ2DQ	Move Quadword to Quadword	3		SSE2	SSE2		
MOVS	Move String	3	Base				
MOVSB	Move String Byte	3	Base				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
MOVSD	Move String Doubleword	3	Base <sup>2</sup>				
MOVSD	Move Scalar Double-Precision Floating-Point	3		SSE2 <sup>2</sup>			
MOVSHDUP	Move Single-Precision High and Duplicate	3		SSE3			
MOVSLDUP	Move Single-Precision Low and Duplicate	3		SSE3			
MOVSQ	Move String Quadword	3	LM				
MOVSS	Move Scalar Single-Precision Floating-Point	3		SSE			
MOVSW	Move String Word	3	Base				
MOVSX	Move with Sign-Extend	3	Base				
MOVSDX	Move with Sign-Extend Doubleword	3	LM				
MOVUPD	Move Unaligned Packed Double-Precision Floating-Point	3		SSE2			
MOVUPS	Move Unaligned Packed Single-Precision Floating-Point	3		SSE			
MOVZX	Move with Zero-Extend	3	Base				
MPSADBW	Multiple Sum of Absolute Differences	3		SSE41			
MUL	Multiply Unsigned	3	Base				
MULPD	Multiply Packed Double-Precision Floating-Point	3		SSE2			
MULPS	Multiply Packed Single-Precision Floating-Point	3		SSE			
MULSD	Multiply Scalar Double-Precision Floating-Point	3		SSE2			
MULSS	Multiply Scalar Single-Precision Floating-Point	3		SSE			
MULX	Multiply Unsigned	3	BMI2				
MWAIT	Monitor Wait <sup>3</sup>	0					MONITOR

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
MWAITX	Monitor Wait with Timeout	3	MONITORX				
NEG	Two's Complement Negation	3	Base				
NOP	No Operation	3	Base				
NOT	One's Complement Negation	3	Base				
OR	Logical OR	3	Base				
ORPD	Logical Bitwise OR Packed Double-Precision Floating-Point	3		SSE2			
ORPS	Logical Bitwise OR Packed Single-Precision Floating-Point	3		SSE			
OUT	Output to Port	3	Base				
OUTS	Output String	3	Base				
OUTSB	Output String Byte	3	Base				
OUTSD	Output String Doubleword	3	Base				
OUTSW	Output String Word	3	Base				
PABSB	Packed Absolute Value Signed Byte	3		SSSE3			
PABSD	Packed Absolute Value Signed Doubleword	3		SSSE3			
PABSW	Packed Absolute Value Signed Word	3		SSSE3			
PACKSSDW	Pack with Saturation Signed Doubleword to Word	3		SSE2	MMX		
PACKSSWB	Pack with Saturation Signed Word to Byte	3		SSE2	MMX		
PACKUSDW	Pack with Unsigned Saturation Doubleword to Word	3		SSE41			
PACKUSWB	Pack with Saturation Signed Word to Unsigned Byte	3		SSE2	MMX		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PADDB	Packed Add Bytes	3		SSE2	MMX		
PADDW	Packed Add Doublewords	3		SSE2	MMX		
PADDQ	Packed Add Quadwords	3		SSE2	SSE2		
PADDSB	Packed Add Signed with Saturation Bytes	3		SSE2	MMX		
PADDSW	Packed Add Signed with Saturation Words	3		SSE2	MMX		
PADDUSB	Packed Add Unsigned with Saturation Bytes	3		SSE2	MMX		
PADDUSW	Packed Add Unsigned with Saturation Words	3		SSE2	MMX		
PADDW	Packed Add Words	3		SSE2	MMX		
PALIGNR	Packed Align Right	3		SSSE3			
PAND	Packed Logical Bitwise AND	3		SSE2	MMX		
PANDN	Packed Logical Bitwise AND NOT	3		SSE2	MMX		
PAUSE	Pause	3	BASE				
PAVGB	Packed Average Unsigned Bytes	3		SSE2	SSE    MmxExt		
PAVGUSB	Packed Average Unsigned Bytes	3			3DNow		
PAVGW	Packed Average Unsigned Words	3		SSE2	SSE    MmxExt		
PBLENDVB	Variable Blend Packed Bytes	3		SSE41			
PBLENDW	Blend Packed Words	3		SSE41			
PCLMULQDQ	Carry-less Multiply Quadwords	3		CLMUL			
PCMPEQB	Packed Compare Equal Bytes	3		SSE2	MMX		
PCMPEQD	Packed Compare Equal Doublewords	3		SSE2	MMX		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PCMPEQQ	Packed Compare Equal Quadwords	3		SSE41			
PCMPEQW	Packed Compare Equal Words	3		SSE2	MMX		
PCMPESTRI	Packed Compare Explicit Length Strings Return Index	3		SSE42			
PCMPESTRM	Packed Compare Explicit Length Strings Return Mask	3		SSE42			
PCMPGTB	Packed Compare Greater Than Signed Bytes	3		SSE2	MMX		
PCMPGTD	Packed Compare Greater Than Signed Doublewords	3		SSE2	MMX		
PCMPGTQ	Packed Compare Greater Than Signed Quadwords	3		SSE42			
PCMPGTW	Packed Compare Greater Than Signed Words	3		SSE2	MMX		
PCMPISTRI	Packed Compare Implicit Length Strings Return Index	3		SSE42			
PCMPISTRM	Packed Compare Implicit Length Strings Return Mask	3		SSE42			
PDEP	Parallel Deposit Bits	3	BMI2				
PEXT	Parallel Extract Bits	3	BMI2				
PEXTRB	Extract Packed Byte	3		SSE41			
PEXTRD	Extract Packed Doubleword	3		SSE41			
PEXTRQ	Extract Packed Quadword	3		SSE41			
PEXTRW	Packed Extract Word	3		SSE41    SSE2	SSE    MmxExt		
PF2ID	Packed Floating-Point to Integer Doubleword Conversion	3			3DNow		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PF2IW	Packed Floating-Point to Integer Word Conversion	3			3DNowExt		
PFACC	Packed Floating-Point Accumulate	3			3DNow		
PFADD	Packed Floating-Point Add	3			3DNow		
PFCMPEQ	Packed Floating-Point Compare Equal	3			3DNow		
PFCMPGE	Packed Floating-Point Compare Greater or Equal	3			3DNow		
PFCMPGT	Packed Floating-Point Compare Greater Than	3			3DNow		
PFMAX	Packed Floating-Point Maximum	3			3DNow		
PFMIN	Packed Floating-Point Minimum	3			3DNow		
PFMUL	Packed Floating-Point Multiply	3			3DNow		
PFNACC	Packed Floating-Point Negative Accumulate	3			3DNowExt		
PFPNACC	Packed Floating-Point Positive-Negative Accumulate	3			3DNowExt		
PFRCPP	Packed Floating-Point Reciprocal Approximation	3			3DNow		
PFRCPI1	Packed Floating-Point Reciprocal, Iteration 1	3			3DNow		
PFRCPI2	Packed Floating-Point Reciprocal or Reciprocal Square Root, Iteration 2	3			3DNow		
PFRSQIT1	Packed Floating-Point Reciprocal Square Root, Iteration 1	3			3DNow		
PFRSQRT	Packed Floating-Point Reciprocal Square Root Approximation	3			3DNow		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PFSUB	Packed Floating-Point Subtract	3			3DNow		
PFSUBR	Packed Floating-Point Subtract Reverse	3			3DNow		
PHADDD	Packed Horizontal Add Doubleword	3		SSSE3			
PHADDSW	Packed Horizontal Add with Saturation Word	3		SSSE3			
PHADDW	Packed Horizontal Add Word	3		SSSE3			
PHMINPOSUW	Horizontal Minimum and Position	3		SSE41			
PHSUBD	Packed Horizontal Subtract Doubleword	3		SSSE3			
PHSUBSW	Packed Horizontal Subtract with Saturation Word	3		SSSE3			
PHSUBW	Packed Horizontal Subtract Word	3		SSSE3			
PI2FD	Packed Integer to Floating-Point Doubleword Conversion	3			3DNow		
PI2FW	Packed Integer To Floating-Point Word Conversion	3			3DNowExt		
PINSRB	Packed Insert Byte	3		SSE41			
PINSRD	Packed Insert Doubleword	3		SSE41			
PINSRQ	Packed Insert Quadword	3		SSE41			
PINSRW	Packed Insert Word	3		SSE2	SSE    MmxExt		
PMADDUBSW	Packed Multiply and Add Unsigned Byte to Signed Word,	3		SSSE3			
PMADDWD	Packed Multiply Words and Add Doublewords	3		SSE2	MMX		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PMAXSB	Packed Maximum Signed Bytes	3		SSE41			
PMAXSD	Packed Maximum Signed Doublewords	3		SSE41			
PMAXSW	Packed Maximum Signed Words	3		SSE2	SSE    MmxExt		
PMAXUB	Packed Maximum Unsigned Bytes	3		SSE2	SSE    MmxExt		
PMAXUD	Packed Maximum Unsigned Doublewords	3		SSE41			
PMAXUW	Packed Maximum Unsigned Words	3		SSE41			
PMINSB	Packed Minimum Signed Bytes	3		SSE41			
PMINSD	Packed Minimum Signed Doublewords	3		SSE41			
PMINSW	Packed Minimum Signed Words	3		SSE2	SSE    MmxExt		
PMINUB	Packed Minimum Unsigned Bytes	3		SSE2	SSE    MmxExt		
PMINUD	Packed Minimum Unsigned Doublewords	3		SSE41			
PMINUW	Packed Minimum Unsigned Words	3		SSE41			
PMOVMSKB	Packed Move Mask Byte	3		SSE2	SSE    MmxExt		
PMOVSXBD	Packed Move with Sign-Extension Byte to Doubleword	3		SSE41			
PMOVSXBQ	Packed Move with Sign Extension Byte to Quadword	3		SSE41			
PMOVSXBW	Packed Move with Sign Extension Byte to Word	3		SSE41			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PMOVSXDW	Packed Move with Sign-Extension Doubleword to Quadword	3		SSE41			
PMOVSXWD	Packed Move with Sign-Extension Word to Doubleword	3		SSE41			
PMOVSXWQ	Packed Move with Sign-Extension Word to Quadword	3		SSE41			
PMOVZXBD	Packed Move with Zero-Extension Byte to Doubleword	3		SSE41			
PMOVZXBQ	Packed Move Byte to Quadword with Zero-Extension	3		SSE41			
PMOVZXBW	Packed Move Byte to Word with Zero-Extension	3		SSE41			
PMOVZXDQ	Packed Move with Zero-Extension Doubleword to Quadword	3		SSE41			
PMOVZXWD	Packed Move Word to Doubleword with Zero-Extension	3		SSE41			
PMOVZXWQ	Packed Move with Zero-Extension Word to Quadword	3		SSE41			
PMULDQ	Packed Multiply Signed Doubleword to Quadword	3		SSE41			
PMULHRSW	Packed Multiply High with Round and Scale Words	3		SSSE3			
PMULHRW	Packed Multiply High Rounded Word	3			3DNow		
PMULHUW	Packed Multiply High Unsigned Word	3		SSE2	SSE    MmxExt		
PMULHW	Packed Multiply High Signed Word	3		SSE2	MMX		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PMULLD	Packed Multiply and Store Low Signed Doubleword	3		SSE41			
PMULLW	Packed Multiply Low Signed Word	3		SSE2	MMX		
PMULUDQ	Packed Multiply Unsigned Doubleword and Store Quadword	3		SSE2	SSE2		
POP	Pop Stack	3	Base				
POPA	Pop All to GPR Words	3	Base				
POPAD	Pop All to GPR Doublewords	3	Base				
POPCNT	Bit Population Count	3	Base				
POPF	Pop to FLAGS Word	3	Base				
POFFD	Pop to EFLAGS Doubleword	3	Base				
POPFQ	Pop to RFLAGS Quadword	3	LM				
POR	Packed Logical Bitwise OR	3		SSE2	MMX		
PREFETCH	Prefetch L1 Data-Cache Line	3	3DNow    3DNowPrefetch    LM				
PREFETCH/level	Prefetch Data to Cache Level <i>level</i>	3	SSE    MmxExt				
PREFETCHW	Prefetch L1 Data-Cache Line for Write	3	3DNow    3DNowPrefetch    LM				
PSADBW	Packed Sum of Absolute Differences of Bytes into a Word	3		SSE2	SSE    MmxExt		
PSHUFB	Packed Shuffle Byte	3		SSSE3			
PSHUFD	Packed Shuffle Doublewords	3		SSE2			
PSHUFHW	Packed Shuffle High Words	3		SSE2			
PSHUFLW	Packed Shuffle Low Words	3		SSE2			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PSHUFW	Packed Shuffle Words	3			SSE    MmxExt		
PSIGNB	Packed Sign Byte	3		SSSE3			
PSIGND	Packed Sign Doubleword	3		SSSE3			
PSIGNW	Packed Sign Word	3		SSSE3			
PSLLD	Packed Shift Left Logical Doublewords	3		SSE2	MMX		
PSLLDQ	Packed Shift Left Logical Double Quadword	3		SSE2			
PSLLQ	Packed Shift Left Logical Quadwords	3		SSE2	MMX		
PSLLW	Packed Shift Left Logical Words	3		SSE2	MMX		
PSRAD	Packed Shift Right Arithmetic Doublewords	3		SSE2	MMX		
PSRAW	Packed Shift Right Arithmetic Words	3		SSE2	MMX		
PSRLD	Packed Shift Right Logical Doublewords	3		SSE2	MMX		
PSRLDQ	Packed Shift Right Logical Double Quadword	3		SSE2			
PSRLQ	Packed Shift Right Logical Quadwords	3		SSE2	MMX		
PSRLW	Packed Shift Right Logical Words	3		SSE2	MMX		
PSUBB	Packed Subtract Bytes	3		SSE2	MMX		
PSUBD	Packed Subtract Doublewords	3		SSE2	MMX		
PSUBQ	Packed Subtract Quadword	3		SSE2	SSE2		
PSUBSB	Packed Subtract Signed With Saturation Bytes	3		SSE2	MMX		
PSUBSW	Packed Subtract Signed with Saturation Words	3		SSE2	MMX		

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PSUBUSB	Packed Subtract Unsigned and Saturate Bytes	3		SSE2	MMX		
PSUBUSW	Packed Subtract Unsigned and Saturate Words	3		SSE2	MMX		
PSUBW	Packed Subtract Words	3		SSE2	MMX		
PSWAPD	Packed Swap Doubleword	3			3DNowExt		
PTEST	Packed Bit Test	3		SSE41			
PUNPCKHBW	Unpack and Interleave High Bytes	3		SSE2	MMX		
PUNPCKHDQ	Unpack and Interleave High Doublewords	3		SSE2	MMX		
PUNPCKHQDQ	Unpack and Interleave High Quadwords	3		SSE2			
PUNPCKHWD	Unpack and Interleave High Words	3		SSE2	MMX		
PUNPCKLBW	Unpack and Interleave Low Bytes	3		SSE2	MMX		
PUNPCKLDQ	Unpack and Interleave Low Doublewords	3		SSE2	MMX		
PUNPCKLQDQ	Unpack and Interleave Low Quadwords	3		SSE2			
PUNPCKLWD	Unpack and Interleave Low Words	3		SSE2	3DNow		
PUSH	Push onto Stack	3	Base				
PUSHA	Push All GPR Words onto Stack	3	Base				
PUSHAD	Push All GPR Doublewords onto Stack	3	Base				
PUSHF	Push EFLAGS Word onto Stack	3	Base				
PUSHFD	Push EFLAGS Doubleword onto Stack	3	Base				
PUSHFQ	Push RFLAGS Quadword onto Stack	3	LM				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
PXOR	Packed Logical Bitwise Exclusive OR	3		SSE2	MMX		
RCL	Rotate Through Carry Left	3	Base				
RCPSS	Reciprocal Packed Single-Precision Floating-Point	3		SSE			
RCPSS	Reciprocal Scalar Single-Precision Floating-Point	3		SSE			
RCR	Rotate Through Carry Right	3	Base				
RDFSBASE	Read FS.base	3					FSGSBASE
RDGSBASE	Read GS.base	3					FSGSBASE
RDMSR	Read Model-Specific Register	0					MSR
RDPMC	Read Performance-Monitoring Counter	3					Base
RDTSC	Read Time-Stamp Counter	3					TSC
RDTSCP	Read Time-Stamp Counter and Processor ID	3					TSC    RDTSCP
RET	Return from Call	3	Base				
ROL	Rotate Left	3	Base				
ROR	Rotate Right	3	Base				
RORX	Rotate Right Extended	3	BMI2				
ROUNDPD	Round Packed Double-Precision Floating-Point	3		SSE41			
ROUNDPS	Round Packed Single-Precision Floating-Point	3		SSE41			
ROUNDSD	Round Scalar Double-Precision Floating-Point	3		SSE41			
ROUNDSS	Round Scalar Single-Precision Floating-Point	3		SSE41			
RSM	Resume from System Management Mode	3					Base

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
RSQRTPS	Reciprocal Square Root Packed Single-Precision Floating-Point	3		SSE			
RSQRTSS	Reciprocal Square Root Scalar Single-Precision Floating-Point	3		SSE			
SAHF	Store AH into Flags	3	LahfSahf				
SAL	Shift Arithmetic Left	3	Base				
SAR	Shift Arithmetic Right	3	Base				
SARX	Shift Right Arithmetic Extended	3	BMI2				
SBB	Subtract with Borrow	3	Base				
SCAS	Scan String	3	Base				
SCASB	Scan String as Bytes	3	Base				
SCASD	Scan String as Doubleword	3	Base				
SCASQ	Scan String as Quadword	3	LM				
SCASW	Scan String as Words	3	Base				
SETcc	Set Byte if Condition	3	Base				
SFENCE	Store Fence	3	SSE    MmxExt				
SGDT	Store Global Descriptor Table Register	3					Base
SHL	Shift Left	3	Base				
SHLD	Shift Left Double	3	Base				
SHLX	Shift Left Logical Extended	3	BMI2				
SHR	Shift Right	3	Base				
SHRD	Shift Right Double	3	Base				
SHRX	Shift Right Logical Extended	3	BMI2				
SHUFPD	Shuffle Packed Double-Precision Floating-Point	3		SSE2			
SHUFPS	Shuffle Packed Single-Precision Floating-Point	3		SSE			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
SIDT	Store Interrupt Descriptor Table Register	3					Base
SKINIT	Secure Init and Jump with Attestation	0					SKINIT
SLDT	Store Local Descriptor Table Register	3					Base
SMSW	Store Machine Status Word	3					Base
SQRTPD	Square Root Packed Double-Precision Floating-Point	3		SSE2			
SQRTPS	Square Root Packed Single-Precision Floating-Point	3		SSE			
SQRTSD	Square Root Scalar Double-Precision Floating-Point	3		SSE2			
SQRTSS	Square Root Scalar Single-Precision Floating-Point	3		SSE			
STC	Set Carry Flag	3	Base				
STD	Set Direction Flag	3	Base				
STGI	Set Global Interrupt Flag	0					SKINIT
STI	Set Interrupt Flag	3					Base
STMXCSR	Store MXCSR Control/Status Register	3		SSE			
STOS	Store String	3	Base				
STOSB	Store String Bytes	3	Base				
STOSD	Store String Doublewords	3	Base				
STOSQ	Store String Quadwords	3	LM				
STOSW	Store String Words	3	Base				
STR	Store Task Register	3					Base
SUB	Subtract	3	Base				
SUBPD	Subtract Packed Double-Precision Floating-Point	3		SSE2			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
SUBPS	Subtract Packed Single-Precision Floating-Point	3		SSE			
SUBSD	Subtract Scalar Double-Precision Floating-Point	3		SSE2			
SUBSS	Subtract Scalar Single-Precision Floating-Point	3		SSE			
SWAPGS	Swap GS Register with KernelGSbase MSR	0					LM
SYSCALL	Fast System Call	3					SYSCALL, SYSRET
SYSENTER	System Call	3					SYS-ENTER, SYSEXIT
SYSEXIT	System Return	0					SYS-ENTER, SYSEXIT
SYSRET	Fast System Return	0					SYSCALL, SYSRET
T1MSKC	Inverse Mask From Trailing Ones	3	TBM				
TEST	Test Bits	3	Base				
TZCNT	Count Trailing Zeros	3	BMI1				
TZMSK	Mask From Trailing Zeros	3	TBM				
UCOMISD	Unordered Compare Scalar Double-Precision Floating-Point	3		SSE2			
UCOMISS	Unordered Compare Scalar Single-Precision Floating-Point	3		SSE			
UD2	Undefined Operation	3					Base
UNPCKHPD	Unpack High Double-Precision Floating-Point	3		SSE2			
UNPCKHPS	Unpack High Single-Precision Floating-Point	3		SSE			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
UNPCKLPD	Unpack Low Double-Precision Floating-Point	3		SSE2			
UNPCKLPS	Unpack Low Single-Precision Floating-Point	3		SSE			
VADDPD	Add Packed Double-Precision Floating-Point	3		AVX			
VADDPS	Add Packed Single-Precision Floating-Point	3		AVX			
VADDSD	Add Scalar Double-Precision Floating-Point	3		AVX			
VADDSS	Add Scalar Single-Precision Floating-Point	3		AVX			
VADDSUBPD	Add and Subtract Double-Precision	3		AVX			
VADDSUBPS	Add and Subtract Single-Precision	3		AVX			
VAESDEC	AES Decryption Round	3		AVX			
VAESDECLAST	AES Last Decryption Round	3		AVX			
VAESENC	AES Encryption Round	3		AVX			
VAESENCLAST	AES Last Encryption Round	3		AVX			
VAESIMC	AES InvMixColumn Transformation	3		AVX			
VAESKEYGENASSIST	AES Assist Round Key Generation	3		AVX			
VANDNPD	AND NOT Packed Double-Precision Floating-Point	3		AVX			
VANDNPS	AND NOT Packed Single-Precision Floating-Point	3		AVX			
VANDPD	AND Packed Double-Precision Floating-Point	3		AVX			
VANDPS	AND Packed Single-Precision Floating-Point	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VBLENDPD	Blend Packed Double-Precision Floating-Point	3		AVX			
VBLENDPS	Blend Packed Single-Precision Floating-Point	3		AVX			
VBLENDVPD	Variable Blend Packed Double-Precision Floating-Point	3		AVX			
VBLENDVPS	Variable Blend Packed Single-Precision Floating-Point	3		AVX			
VBROADCASTF128	Load With Broadcast From 128-bit Memory Location	3		AVX			
VBROADCASTI128	Load With Broadcast Integer From 128-bit Memory Location	3		AVX2			
VBROADCASTSD	Load With Broadcast From 64-Bit Memory Location	3		AVX, AVX2 <sup>4</sup>			
VBROADCASTSS	Load With Broadcast From 32-Bit Memory Location	3		AVX, AVX2 <sup>4</sup>			
VCMPPD	Compare Packed Double-Precision Floating-Point	3		AVX			
VCMPPS	Compare Packed Single-Precision Floating-Point	3		AVX			
VCMPSD	Compare Scalar Double-Precision Floating-Point	3		AVX			
VCMPSS	Compare Scalar Single-Precision Floating-Point	3		AVX			
VCOMISD	Compare Ordered Scalar Double-Precision Floating-Point	3		AVX			
VCOMISS	Compare Ordered Scalar Single-Precision Floating-Point	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VCVTDQ2PD	Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point	3		AVX			
VCVTDQ2PS	Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point	3		AVX			
VCVTPD2DQ	Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers	3		AVX			
VCVTPD2PS	Convert Packed Double-Precision Floating-Point to Packed Single-Precision Floating-Point	3		AVX			
VCVTPH2PS	Convert Packed 16-Bit Floating-Point to Single-Precision Floating-Point	3		AVX			
VCVTPS2DQ	Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers	3		AVX			
VCVTPS2PD	Convert Packed Single-Precision Floating-Point to Packed Double-Precision Floating-Point	3		AVX			
VCVTPS2PH	Convert Packed Single-Precision Floating-Point to 16-Bit Floating-Point	3		AVX			
VCVTSD2SI	Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VCVTSD2SS	Convert Scalar Double-Precision Floating-Point to Scalar Single-Precision Floating-Point	3		AVX			
VCVTSI2SD	Convert Signed Doubleword or Quadword Integer to Scalar Double-Precision Floating-Point	3		AVX			
VCVTSI2SS	Convert Signed Doubleword or Quadword Integer to Scalar Single-Precision Floating-Point	3		AVX			
VCVTSS2SD	Convert Scalar Single-Precision Floating-Point to Scalar Double-Precision Floating-Point	3		AVX			
VCVTSS2SI	Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer	3		AVX			
VCVTPD2DQ	Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated	3		AVX			
VCVTPS2DQ	Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated	3		AVX			
VCVTTSD2SI	Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VCVTTSS2SI	Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated	3		AVX			
VDIVPD	Divide Packed Double-Precision Floating-Point	3		AVX			
VDIVPS	Divide Packed Single-Precision Floating-Point	3		AVX			
VDIVSD	Divide Scalar Double-Precision Floating-Point	3		AVX			
VDIVSS	Divide Scalar Single-Precision Floating-Point	3		AVX			
VDPPD	Dot Product Packed Double-Precision Floating-Point	3		AVX			
VDPPS	Dot Product Packed Single-Precision Floating-Point	3		AVX			
VERR	Verify Segment for Reads	3					Base
VERW	Verify Segment for Writes	3					Base
VEXTRACTF128	Extract Packed Values from 128-bit Memory Location	3		AVX			
VEXTRACTI128	Extract 128-bit Integer	3		AVX2			
VEXTRACTPS	Extract Packed Single-Precision Floating-Point	3		AVX			
VFMADDPD	Multiply and Add Packed Double-Precision Floating-Point	3		FMA4			
VFMADD132PD	Multiply and Add Packed Double-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFMADD213PD	Multiply and Add Packed Double-Precision Floating-Point	3		FMA			
VFMADD231PD	Multiply and Add Packed Double-Precision Floating-Point	3		FMA			
VFMADDPS	Multiply and Add Packed Single-Precision Floating-Point	3		FMA4			
VFMADD132PS	Multiply and Add Packed Single-Precision Floating-Point	3		FMA			
VFMADD213PS	Multiply and Add Packed Single-Precision Floating-Point	3		FMA			
VFMADD231PS	Multiply and Add Packed Single-Precision Floating-Point	3		FMA			
VFMADDSD	Multiply and Add Scalar Double-Precision Floating-Point	3		FMA4			
VFMADD132SD	Multiply and Add Scalar Double-Precision Floating-Point	3		FMA			
VFMADD213SD	Multiply and Add Scalar Double-Precision Floating-Point	3		FMA			
VFMADD231SD	Multiply and Add Scalar Double-Precision Floating-Point	3		FMA			
VFMADDSS	Multiply and Add Scalar Single-Precision Floating-Point	3		FMA4			
VFMADD132SS	Multiply and Add Scalar Single-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFMADD213SS	Multiply and Add Scalar Single-Precision Floating-Point	3		FMA			
VFMADD231SS	Multiply and Add Scalar Single-Precision Floating-Point	3		FMA			
VFMADDSUBPD	Multiply with Alternating Add/Subtract Packed Double-Precision Floating-Point	3		FMA4			
VFMADDSUB132PD	Multiply with Alternating Add/Subtract Packed Double-Precision Floating-Point	3		FMA			
VFMADDSUB213PD	Multiply with Alternating Add/Subtract Packed Double-Precision Floating-Point	3		FMA			
VFMADDSUB231PD	Multiply with Alternating Add/Subtract Packed Double-Precision Floating-Point	3		FMA			
VFMADDSUBPS	Multiply with Alternating Add/Subtract Packed Single-Precision Floating-Point	3		FMA4			
VFMADDSUB132PS	Multiply with Alternating Add/Subtract Packed Single-Precision Floating-Point	3		FMA			
VFMADDSUB213PS	Multiply with Alternating Add/Subtract Packed Single-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFMADDSSUB231PS	Multiply with Alternating Add/Subtract Packed Single-Precision Floating-Point	3		FMA			
VFMSUBADDPPD	Multiply with Alternating Subtract/Add Packed Double-Precision Floating-Point	3		FMA4			
VFMSUBADD132PD	Multiply with Alternating Subtract/Add Packed Double-Precision Floating-Point	3		FMA			
VFMSUBADD213PD	Multiply with Alternating Subtract/Add Packed Double-Precision Floating-Point	3		FMA			
VFMSUBADD231PD	Multiply with Alternating Subtract/Add Packed Double-Precision Floating-Point	3		FMA			
VFMSUBADDPS	Multiply with Alternating Subtract/Add Packed Single-Precision Floating-Point	3		FMA4			
VFMSUBADD132PS	Multiply with Alternating Subtract/Add Packed Single-Precision Floating-Point	3		FMA			
VFMSUBADD213PS	Multiply with Alternating Subtract/Add Packed Single-Precision Floating-Point	3		FMA			
VFMSUBADD231PS	Multiply with Alternating Subtract/Add Packed Single-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFMSUBPD	Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA4			
VFMSUB132PD	Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA			
VFMSUB213PD	Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA			
VFMSUB231PD	Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA			
VFMSUBPS	Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA4			
VFMSUB132PS	Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA			
VFMSUB213PS	Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA			
VFMSUB231PS	Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA			
VFMSUBSD	Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA4			
VFMSUB132SD	Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA			
VFMSUB213SD	Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA			
VFMSUB231SD	Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFMSUBSS	Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA4			
VFMSUB132SS	Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA			
VFMSUB213SS	Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA			
VFMSUB231SS	Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA			
VFNMADDPD	Negative Multiply and Add Packed Double-Precision Floating-Point	3		FMA4			
VFNMADD132PD	Negative Multiply and Add Packed Double-Precision Floating-Point	3		FMA			
VFNMADD213PD	Negative Multiply and Add Packed Double-Precision Floating-Point	3		FMA			
VFNMADD231PD	Negative Multiply and Add Packed Double-Precision Floating-Point	3		FMA			
VFNMADDPD	Negative Multiply and Add Packed Single-Precision Floating-Point	3		FMA4			
VFNMADD132PS	Negative Multiply and Add Packed Single-Precision Floating-Point	3		FMA			
VFNMADD213PS	Negative Multiply and Add Packed Single-Precision Floating-Point	3		FMA			
VFNMADD231PS	Negative Multiply and Add Packed Single-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFNMADDSD	Negative Multiply and Add Scalar Double-Precision Floating-Point	3		FMA4			
VFNMADD132SD	Negative Multiply and Add Scalar Double-Precision Floating-Point	3		FMA			
VFNMADD213SD	Negative Multiply and Add Scalar Double-Precision Floating-Point	3		FMA			
VFNMADD231SD	Negative Multiply and Add Scalar Double-Precision Floating-Point	3		FMA			
VFNMADDSS	Negative Multiply and Add Scalar Single-Precision Floating-Point	3		FMA4			
VFNMADD132SS	Negative Multiply and Add Scalar Single-Precision Floating-Point	3		FMA			
VFNMADD213SS	Negative Multiply and Add Scalar Single-Precision Floating-Point	3		FMA			
VFNMADD231SS	Negative Multiply and Add Scalar Single-Precision Floating-Point	3		FMA			
VFNMSUBPD	Negative Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA4			
VFNMSUB132PD	Negative Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA			
VFNMSUB213PD	Negative Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA			
VFNMSUB231PD	Negative Multiply and Subtract Packed Double-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFNMSUBPS	Negative Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA4			
VFNMSUB132PS	Negative Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA			
VFNMSUB213PS	Negative Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA			
VFNMSUB231PS	Negative Multiply and Subtract Packed Single-Precision Floating-Point	3		FMA			
VFNMSUBSD	Negative Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA4			
VFNMSUB132SD	Negative Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA			
VFNMSUB213SD	Negative Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA			
VFNMSUB231SD	Negative Multiply and Subtract Scalar Double-Precision Floating-Point	3		FMA			
VFNMSUBSS	Negative Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA4			
VFNMSUB132SS	Negative Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA			
VFNMSUB213SS	Negative Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA			
VFNMSUB231SS	Negative Multiply and Subtract Scalar Single-Precision Floating-Point	3		FMA			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VFRCZPD	Extract Fraction Packed Double-Precision Floating-Point	3		XOP			
VFRCZPS	Extract Fraction Packed Single-Precision Floating-Point	3		XOP			
VFRCZSD	Extract Fraction Scalar Double-Precision Floating-Point	3		XOP			
VFRCZSS	Extract Fraction Scalar Single-Precision Floating Point	3		XOP			
VGATHERDPD	Conditionally Gather Double-Precision Floating-Point Values, Doubleword Indices	3		AVX2			
VGATHERDPS	Conditionally Gather Single-Precision Floating-Point Values, Doubleword Indices	3		AVX2			
VGATHERQPD	Conditionally Gather Double-Precision Floating-Point Values, Quadword Indices	3		AVX2			
VGATHERQPS	Conditionally Gather Single-Precision Floating-Point Values, Quadword Indices	3		AVX2			
VHADDPD	Horizontal Add Packed Double	3		AVX			
VHADDPS	Horizontal Add Packed Single	3		AVX			
VHSUBPD	Horizontal Subtract Packed Double	3		AVX			
VHSUBPS	Horizontal Subtract Packed Single	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VINSETRF128	Insert Packed Values 128-bit	3		AVX			
VINSETI128	Insert Packed Integer Values 128-bit	3		AVX2			
VINSETPS	Insert Packed Single-Precision Floating-Point	3		AVX			
VLDDQU	Load Unaligned Double Quadword	3		AVX			
VLDMXCSR	Load MXCSR Control/Status Register	3		AVX			
VMASKMOVDQU	Masked Move Double Quadword Unaligned	3		AVX			
VMASKMOVPD	Masked Move Packed Double-Precision	3		AVX			
VMASKMOVPS	Masked Move Packed Single-Precision	3		AVX			
VMAXPD	Maximum Packed Double-Precision Floating-Point	3		AVX			
VMAXPS	Maximum Packed Single-Precision Floating-Point	3		AVX			
VMAXSD	Maximum Scalar Double-Precision Floating-Point	3		AVX			
VMAXSS	Maximum Scalar Single-Precision Floating-Point	3		AVX			
VMINPD	Minimum Packed Double-Precision Floating-Point	3		AVX			
VMINPS	Minimum Packed Single-Precision Floating-Point	3		AVX			
VMINSD	Minimum Scalar Double-Precision Floating-Point	3		AVX			
VMINSS	Minimum Scalar Single-Precision Floating-Point	3		AVX			
VMOVAPD	Move Aligned Packed Double-Precision Floating-Point	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VMOVAPS	Move Aligned Packed Single-Precision Floating-Point	3		AVX			
VMOVD	Move Doubleword or Quadword	3		AVX			
VMOVDDUP	Move Double-Precision and Duplicate	3		AVX			
VMOVDQA	Move Aligned Double Quadword	3		AVX			
VMOVDQU	Move Unaligned Double Quadword	3		AVX			
VMOVHLPS	Move Packed Single-Precision Floating-Point High to Low	3		AVX			
VMOVHPD	Move High Packed Double-Precision Floating-Point	3		AVX			
VMOVHPS	Move High Packed Single-Precision Floating-Point	3		AVX			
VMOVLHPS	Move Packed Single-Precision Floating-Point Low to High	3		AVX			
VMOVLPD	Move Low Packed Double-Precision Floating-Point	3		AVX			
VMOVLPD	Move Low Packed Single-Precision Floating-Point	3		AVX			
VMOVMSKPD	Extract Packed Double-Precision Floating-Point Sign Mask	3		AVX			
VMOVMSKPS	Extract Packed Single-Precision Floating-Point Sign Mask	3		AVX			
VMOVNTDQ	Move Non-Temporal Double Quadword	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VMOVNTDQA	Move Non-Temporal Double Quadword Aligned	3		AVX, AVX2 <sup>4</sup>			
VMOVNTPD	Move Non-Temporal Packed Double-Precision Floating-Point	3		AVX			
VMOVNTPS	Move Non-Temporal Packed Single-Precision Floating-Point	3		AVX			
VMOVQ	Move Quadword	3		AVX			
VMOVS D	Move Scalar Double-Precision Floating-Point	3		AVX			
VMOVSHDUP	Move Single-Precision High and Duplicate	3		AVX			
VMOVSLDUP	Move Single-Precision Low and Duplicate	3		AVX			
VMOVSS	Move Scalar Single-Precision Floating-Point	3		AVX			
VMOVUPD	Move Unaligned Packed Double-Precision Floating-Point	3		AVX			
VMOVUPS	Move Unaligned Packed Single-Precision Floating-Point	3		AVX			
VMPSADBW	Multiple Sum of Absolute Differences	3		AVX, AVX2 <sup>4</sup>			
VMULPD	Multiply Packed Double-Precision Floating-Point	3		AVX			
VMULPS	Multiply Packed Single-Precision Floating-Point	3		AVX			
VMULSD	Multiply Scalar Double-Precision Floating-Point	3		AVX			
VMULSS	Multiply Scalar Single-Precision Floating-Point	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VORPD	Logical Bitwise OR Packed Double-Precision Floating-Point	3		AVX			
VORPS	Logical Bitwise OR Packed Single-Precision Floating-Point	3		AVX			
VPABSB	Packed Absolute Value Signed Byte	3		AVX, AVX2 <sup>4</sup>			
VPABSD	Packed Absolute Value Signed Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPABSW	Packed Absolute Value Signed Word	3		AVX, AVX2 <sup>4</sup>			
VPACKSSDW	Pack with Saturation Signed Doubleword to Word	3		AVX, AVX2 <sup>4</sup>			
VPACKSSWB	Pack with Saturation Signed Word to Byte	3		AVX, AVX2 <sup>4</sup>			
VPACKUSDW	Pack with Unsigned Saturation Doubleword to Word	3		AVX, AVX2 <sup>4</sup>			
VPACKUSWB	Pack with Saturation Signed Word to Unsigned Byte	3		AVX, AVX2 <sup>4</sup>			
VPADDB	Packed Add Bytes	3		AVX, AVX2 <sup>4</sup>			
VPADDD	Packed Add Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPADDQ	Packed Add Quadwords	3		AVX, AVX2 <sup>4</sup>			
VPADDSB	Packed Add Signed with Saturation Bytes	3		AVX, AVX2 <sup>4</sup>			
VPADDSW	Packed Add Signed with Saturation Words	3		AVX, AVX2 <sup>4</sup>			
VPADDUSB	Packed Add Unsigned with Saturation Bytes	3		AVX, AVX2 <sup>4</sup>			
VPADDUSW	Packed Add Unsigned with Saturation Words	3		AVX, AVX2 <sup>4</sup>			
VPADDW	Packed Add Words	3		AVX, AVX2 <sup>4</sup>			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPALIGNR	Packed Align Right	3		AVX, AVX2 <sup>4</sup>			
VPAND	Packed Logical Bitwise AND	3		AVX, AVX2 <sup>4</sup>			
VPANDN	Packed Logical Bitwise AND NOT	3		AVX, AVX2 <sup>4</sup>			
VPAVGB	Packed Average Unsigned Bytes	3		AVX, AVX2 <sup>4</sup>			
VPAVGW	Packed Average Unsigned Words	3		AVX, AVX2 <sup>4</sup>			
VPBLEND <sup>D</sup>	Blend Packed Doublewords	3		AVX2			
VPBLENDVB	Variable Blend Packed Bytes	3		AVX, AVX2 <sup>4</sup>			
VPBLENDW	Blend Packed Words	3		AVX, AVX2 <sup>4</sup>			
VPBROADCASTB	Broadcast Packed Byte	3		AVX2			
VPBROADCASTD	Broadcast Packed Doubleword	3		AVX2			
VPBROADCASTQ	Broadcast Packed Quadword	3		AVX2			
VPBROADCASTW	Broadcast Packed Word	3		AVX2			
VPCLMULQDQ	Carry-less Multiply Quadwords	3		CLMUL    AVX			
VPCM <sup>M</sup> OV	Vector Conditional Move	3		XOP			
VPCMPEQB	Packed Compare Equal Bytes	3		AVX, AVX2 <sup>4</sup>			
VPCMPEQD	Packed Compare Equal Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPCMPEQQ	Packed Compare Equal Quadwords	3		AVX, AVX2 <sup>4</sup>			
VPCMPEQW	Packed Compare Equal Words	3		AVX, AVX2 <sup>4</sup>			
VPCMPESTRI	Packed Compare Explicit Length Strings Return Index	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPCMPESTRM	Packed Compare Explicit Length Strings Return Mask	3		AVX			
VPCMPGTB	Packed Compare Greater Than Signed Bytes	3		AVX, AVX2 <sup>4</sup>			
VPCMPGTD	Packed Compare Greater Than Signed Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPCMPGTQ	Packed Compare Greater Than Signed Quadwords	3		AVX, AVX2 <sup>4</sup>			
VPCMPGTW	Packed Compare Greater Than Signed Words	3		AVX, AVX2 <sup>4</sup>			
VPCMPISTRI	Packed Compare Implicit Length Strings Return Index	3		AVX			
VPCMPISTRM	Packed Compare Implicit Length Strings Return Mask	3		AVX			
VPCOMB	Compare Vector Signed Bytes	3		XOP			
VPCOMD	Compare Vector Signed Doublewords	3		XOP			
VPCOMQ	Compare Vector Signed Quadwords	3		XOP			
VPCOMUB	Compare Vector Unsigned Bytes	3		XOP			
VPCOMUD	Compare Vector Unsigned Doublewords	3		XOP			
VPCOMUQ	Compare Vector Unsigned Quadwords	3		XOP			
VPCOMUW	Compare Vector Unsigned Words	3		XOP			
VPCOMW	Compare Vector Signed Words	3		XOP			
VPERM2F128	Permute Floating-Point 128-bit	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPERM2I128	Permute Integer 128-bit	3		AVX2			
VPERMD	Packed Permute Doubleword	3		AVX2			
VPERMIL2PD	Permute Two-Source Double-Precision Floating-Point	3		XOP			
VPERMIL2PS	Permute Two-Source Single-Precision Floating-Point	3		XOP			
VPERMILPD	Permute Double-Precision	3		AVX			
VPERMILPS	Permute Single-Precision	3		AVX			
VPERMPD	Packed Permute Double-Precision Floating-Point	3		AVX2			
VPERMPS	Packed Permute Single-Precision Floating-Point	3		AVX2			
VPERMQ	Packed Permute Quadword	3		AVX2			
VPEXTRB	Extract Packed Byte	3		AVX			
VPEXTRD	Extract Packed Doubleword	3		AVX			
VPEXTRQ	Extract Packed Quadword	3		AVX			
VPEXTRW	Packed Extract Word	3		AVX			
VPGATHERDD	Conditionally Gather Doublewords, Doubleword Indices	3		AVX2			
VPGATHERDQ	Conditionally Gather Quadwords, Doubleword Indices	3		AVX2			
VPGATHERQD	Conditionally Gather Doublewords, Quadword Indices	3		AVX2			
VPGATHERQQ	Conditionally Gather Quadwords, Quadword Indices	3		AVX2			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPHADDBD	Packed Horizontal Add Signed Byte to Signed Doubleword	3		XOP			
VPHADDBQ	Packed Horizontal Add Signed Byte to Signed Quadword	3		XOP			
VPHADDBW	Packed Horizontal Add Signed Byte to Signed Word	3		XOP			
VPHADD	Packed Horizontal Add Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPHADDDQ	Packed Horizontal Add Signed Doubleword to Signed Quadword	3		XOP			
VPHADDSW	Packed Horizontal Add with Saturation Word	3		AVX, AVX2 <sup>4</sup>			
VPHADDUBD	Packed Horizontal Add Unsigned Byte to Doubleword	3		XOP			
VPHADDUBQ	Packed Horizontal Add Unsigned Byte to Quadword	3		XOP			
VPHADDUBW	Packed Horizontal Add Unsigned Byte to Word	3		XOP			
VPHADDUDQ	Packed Horizontal Add Unsigned Doubleword to Quadword	3		XOP			
VPHADDUWD	Packed Horizontal Add Unsigned Word to Doubleword	3		XOP			
VPHADDUWQ	Packed Horizontal Add Unsigned Word to Quadword	3		XOP			
VPHADDW	Packed Horizontal Add Word	3		AVX, AVX2 <sup>4</sup>			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPHADDWD	Packed Horizontal Add Signed Word to Signed Doubleword	3		XOP			
VPHADDWQ	Packed Horizontal Add Signed Word to Signed Quadword	3		XOP			
VPHMINPOSUW	Horizontal Minimum and Position	3		AVX			
VPHSUBBW	Packed Horizontal Subtract Signed Byte to Signed Word	3		XOP			
VPHSUBD	Packed Horizontal Subtract Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPHSUBDQ	Packed Horizontal Subtract Signed Doubleword to Signed Quadword	3		XOP			
VPHSUBSW	Packed Horizontal Subtract with Saturation Word	3		AVX, AVX2 <sup>4</sup>			
VPHSUBW	Packed Horizontal Subtract Word	3		AVX, AVX2 <sup>4</sup>			
VPHSUBWD	Packed Horizontal Subtract Signed Word to Signed Doubleword	3		XOP			
VPINSRB	Packed Insert Byte	3		AVX			
VPINSRD	Packed Insert Doubleword	3		AVX			
VPINSRQ	Packed Insert Quadword	3		AVX			
VPINSRW	Packed Insert Word	3		AVX			
VPMACSD	Packed Multiply Accumulate Signed Doubleword to Signed Doubleword	3		XOP			
VPMACSDQH	Packed Multiply Accumulate Signed High Doubleword to Signed Quadword	3		XOP			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPMACSDQL	Packed Multiply Accumulate Signed Low Doubleword to Signed Quadword	3		XOP			
VPMACSSDD	Packed Multiply Accumulate with Saturation Signed Doubleword to Signed Doubleword	3		XOP			
VPMACSSDQH	Packed Multiply Accumulate with Saturation Signed High Doubleword to Signed Quadword	3		XOP			
VPMACSSDQL	Packed Multiply Accumulate with Saturation Signed Low Doubleword to Signed Quadword	3		XOP			
VPMACSSWD	Packed Multiply Accumulate with Saturation Signed Word to Signed Doubleword	3		XOP			
VPMACSSWW	Packed Multiply Accumulate with Saturation Signed Word to Signed Word	3		XOP			
VPMACSWD	Packed Multiply Accumulate Signed Word to Signed Doubleword	3		XOP			
VPMACSWW	Packed Multiply Accumulate Signed Word to Signed Word	3		XOP			
VPMADCSSWD	Packed Multiply Add Accumulate with Saturation Signed Word to Signed Doubleword	3		XOP			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPMADCSWD	Packed Multiply Add Accumulate Signed Word to Signed Doubleword	3		XOP			
VPMADDUBSW	Packed Multiply and Add Unsigned Byte to Signed Word	3		AVX, AVX2 <sup>4</sup>			
VPMADDWD	Packed Multiply Words and Add Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPMASKMOVD	Masked Move Packed Doubleword	3		AVX2			
VPMASKMOVQ	Masked Move Packed Quadword	3		AVX2			
VPMAXSB	Packed Maximum Signed Bytes	3		AVX, AVX2 <sup>4</sup>			
VPMAXSD	Packed Maximum Signed Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPMAXSW	Packed Maximum Signed Words	3		AVX, AVX2 <sup>4</sup>			
VPMAXUB	Packed Maximum Unsigned Bytes	3		AVX, AVX2 <sup>4</sup>			
VPMAXUD	Packed Maximum Unsigned Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPMAXUW	Packed Maximum Unsigned Words	3		AVX, AVX2 <sup>4</sup>			
VPMINSB	Packed Minimum Signed Bytes	3		AVX, AVX2 <sup>4</sup>			
VPMINSD	Packed Minimum Signed Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPMINSW	Packed Minimum Signed Words	3		AVX, AVX2 <sup>4</sup>			
VPMINUB	Packed Minimum Unsigned Bytes	3		AVX, AVX2 <sup>4</sup>			
VPMINUD	Packed Minimum Unsigned Doublewords	3		AVX, AVX2 <sup>4</sup>			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPMINUW	Packed Minimum Unsigned Words	3		AVX, AVX2 <sup>4</sup>			
VPMOVMSKB	Packed Move Mask Byte	3		AVX, AVX2 <sup>4</sup>			
VPMOVSXBD	Packed Move with Sign-Extension Byte to Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPMOVSXBQ	Packed Move with Sign Extension Byte to Quadword	3		AVX, AVX2 <sup>4</sup>			
VPMOVSXBW	Packed Move with Sign Extension Byte to Word	3		AVX, AVX2 <sup>4</sup>			
VPMOVSXDQ	Packed Move with Sign-Extension Doubleword to Quadword	3		AVX, AVX2 <sup>4</sup>			
VPMOVSXWD	Packed Move with Sign-Extension Word to Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPMOVSXWQ	Packed Move with Sign-Extension Word to Quadword	3		AVX, AVX2 <sup>4</sup>			
VPMOVZXBD	Packed Move with Zero-Extension Byte to Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPMOVZXBQ	Packed Move Byte to Quadword with Zero-Extension	3		AVX, AVX2 <sup>4</sup>			
VPMOVZXBW	Packed Move Byte to Word with Zero-Extension	3		AVX, AVX2 <sup>4</sup>			
VPMOVZXDQ	Packed Move with Zero-Extension Doubleword to Quadword	3		AVX, AVX2 <sup>4</sup>			
VPMOVZXWD	Packed Move Word to Doubleword with Zero-Extension	3		AVX, AVX2 <sup>4</sup>			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>					
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System	
VPMOVZXWQ	Packed Move with Zero-Extension Word to Quadword	3		AVX, AVX2 <sup>4</sup>				
VPMULDQ	Packed Multiply Signed Doubleword to Quadword	3		AVX, AVX2 <sup>4</sup>				
VPMULHRSW	Packed Multiply High with Round and Scale Words	3		AVX, AVX2 <sup>4</sup>				
VPMULHUW	Packed Multiply High Unsigned Word	3		AVX, AVX2 <sup>4</sup>				
VPMULHW	Packed Multiply High Signed Word	3		AVX, AVX2 <sup>4</sup>				
VPMULLD	Packed Multiply and Store Low Signed Doubleword	3		AVX, AVX2 <sup>4</sup>				
VPMULLW	Packed Multiply Low Signed Word	3		AVX, AVX2 <sup>4</sup>				
VPMULUDQ	Packed Multiply Unsigned Doubleword and Store Quadword	3		AVX, AVX2 <sup>4</sup>				
VPOR	Packed Logical Bitwise OR	3		AVX, AVX2 <sup>4</sup>				
VPPERM	Packed Permute Bytes	3		XOP				
VPROTB	Packed Rotate Bytes	3		XOP				
VPROTD	Packed Rotate Doublewords	3		XOP				
VPROTQ	Packed Rotate Quadwords	3		XOP				
VPROTW	Packed Rotate Words	3		XOP				
VPSADBW	Packed Sum of Absolute Differences of Bytes into a Word	3		AVX, AVX2 <sup>4</sup>				
VPSHAB	Packed Shift Arithmetic Bytes	3		XOP				
VPSHAD	Packed Shift Arithmetic Doublewords	3		XOP				
VPSHAQ	Packed Shift Arithmetic Quadwords	3		XOP				

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPSHAW	Packed Shift Arithmetic Words	3		XOP			
VPSHLB	Packed Shift Logical Bytes	3		XOP			
VPSHLD	Packed Shift Logical Doublewords	3		XOP			
VPSHLQ	Packed Shift Logical Quadwords	3		XOP			
VPSHLW	Packed Shift Logical Words	3		XOP			
VPSHUFB	Packed Shuffle Byte	3		AVX, AVX2 <sup>4</sup>			
VPSHUFD	Packed Shuffle Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPSHUFHW	Packed Shuffle High Words	3		AVX, AVX2 <sup>4</sup>			
VPSHUFLW	Packed Shuffle Low Words	3		AVX, AVX2 <sup>4</sup>			
VPSIGNB	Packed Sign Byte	3		AVX, AVX2 <sup>4</sup>			
VPSIGND	Packed Sign Doubleword	3		AVX, AVX2 <sup>4</sup>			
VPSIGNW	Packed Sign Word	3		AVX, AVX2 <sup>4</sup>			
VPSLLD	Packed Shift Left Logical Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPSLLDQ	Packed Shift Left Logical Double Quadword	3		AVX, AVX2 <sup>4</sup>			
VPSLLQ	Packed Shift Left Logical Quadwords	3		AVX, AVX2 <sup>4</sup>			
VPSLLVD	Variable Shift Left Logical Doublewords	3		AVX2			
VPSLLVQ	Variable Shift Left Logical Quadwords	3		AVX2			
VPSLLW	Packed Shift Left Logical Words	3		AVX, AVX2 <sup>4</sup>			
VPSRAD	Packed Shift Right Arithmetic Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPSRAVD	Variable Shift Right Arithmetic Doublewords	3		AVX2			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPSRAW	Packed Shift Right Arithmetic Words	3		AVX, AVX2 <sup>4</sup>			
VPSRLD	Packed Shift Right Logical Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPSRLDQ	Packed Shift Right Logical Double Quadword	3		AVX, AVX2 <sup>4</sup>			
VPSRLQ	Packed Shift Right Logical Quadwords	3		AVX, AVX2 <sup>4</sup>			
VPSRLVD	Variable Shift Right Logical Doublewords	3		AVX2			
VPSRLVQ	Variable Shift Right Logical Quadwords	3		AVX2			
VPSRLW	Packed Shift Right Logical Words	3		AVX, AVX2 <sup>4</sup>			
VPSUBB	Packed Subtract Bytes	3		AVX, AVX2 <sup>4</sup>			
VPSUBD	Packed Subtract Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPSUBQ	Packed Subtract Quadword	3		AVX, AVX2 <sup>4</sup>			
VPSUBSB	Packed Subtract Signed With Saturation Bytes	3		AVX, AVX2 <sup>4</sup>			
VPSUBSW	Packed Subtract Signed with Saturation Words	3		AVX, AVX2 <sup>4</sup>			
VPSUBUSB	Packed Subtract Unsigned and Saturate Bytes	3		AVX, AVX2 <sup>4</sup>			
VPSUBUSW	Packed Subtract Unsigned and Saturate Words	3		AVX, AVX2 <sup>4</sup>			
VPSUBW	Packed Subtract Words	3		AVX, AVX2 <sup>4</sup>			
VPTEST	Packed Bit Test	3		AVX			
VPUNPCKHBW	Unpack and Interleave High Bytes	3		AVX, AVX2 <sup>4</sup>			
VPUNPCKHDQ	Unpack and Interleave High Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPUNPCKHQDQ	Unpack and Interleave High Quadwords	3		AVX, AVX2 <sup>4</sup>			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VPUNPCKHWD	Unpack and Interleave High Words	3		AVX, AVX2 <sup>4</sup>			
VPUNPCKLBW	Unpack and Interleave Low Bytes	3		AVX, AVX2 <sup>4</sup>			
VPUNPCKLDQ	Unpack and Interleave Low Doublewords	3		AVX, AVX2 <sup>4</sup>			
VPUNPCKLQDQ	Unpack and Interleave Low Quadwords	3		AVX, AVX2 <sup>4</sup>			
VPUNPCKLWD	Unpack and Interleave Low Words	3		AVX, AVX2 <sup>4</sup>			
VPXOR	Packed Logical Bitwise Exclusive OR	3		AVX, AVX2 <sup>4</sup>			
VRCPPS	Reciprocal Packed Single-Precision Floating-Point	3		AVX			
VRCPSS	Reciprocal Scalar Single-Precision Floating-Point	3		AVX			
VROUNDPD	Round Packed Double-Precision Floating-Point	3		AVX			
VROUNDPS	Round Packed Single-Precision Floating-Point	3		AVX			
VROUNDSD	Round Scalar Double-Precision Floating-Point	3		AVX			
VROUNDSS	Round Scalar Single-Precision Floating-Point	3		AVX			
VRSQRTPS	Reciprocal Square Root Packed Single-Precision Floating-Point	3		AVX			
VRSQRTSS	Reciprocal Square Root Scalar Single-Precision Floating-Point	3		AVX			
VSHUFPD	Shuffle Packed Double-Precision Floating-Point	3		AVX			
VSHUFPS	Shuffle Packed Single-Precision Floating-Point	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flags(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VSQRTPD	Square Root Packed Double-Precision Floating-Point	3		AVX			
VSQRTPS	Square Root Packed Single-Precision Floating-Point	3		AVX			
VSQRTSD	Square Root Scalar Double-Precision Floating-Point	3		AVX			
VSQRTSS	Square Root Scalar Single-Precision Floating-Point	3		AVX			
VSTMXCSR	Store MXCSR Control/Status Register	3		AVX			
VSUBPD	Subtract Packed Double-Precision Floating-Point	3		AVX			
VSUBPS	Subtract Packed Single-Precision Floating-Point	3		AVX			
VSUBSD	Subtract Scalar Double-Precision Floating-Point	3		AVX			
VSUBSS	Subtract Scalar Single-Precision Floating-Point	3		AVX			
VMLOAD	Load State from VMCB	0					SVM
VMMCALL	Call VMM	0					SVM
VMRUN	Run Virtual Machine	0					SVM
VMSAVE	Save State to VMCB	0					SVM
VTESTPD	Packed Bit Test	3		AVX			
VTESTPS	Packed Bit Test	3		AVX			
VUCOMISD	Unordered Compare Scalar Double-Precision Floating-Point	3		AVX			
VUCOMISS	Unordered Compare Scalar Single-Precision Floating-Point	3		AVX			
VUNPCKHPD	Unpack High Double-Precision Floating-Point	3		AVX			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
VUNPCKHPS	Unpack High Single-Precision Floating-Point	3		AVX			
VUNPCKLPD	Unpack Low Double-Precision Floating-Point	3		AVX			
VUNPCKLPS	Unpack Low Single-Precision Floating-Point	3		AVX			
VXORPD	Logical Bitwise Exclusive OR Packed Double-Precision Floating-Point	3		AVX			
VXORPS	Logical Bitwise Exclusive OR Packed Single-Precision Floating-Point	3		AVX			
VZEROALL	Zero All YMM Registers	3		AVX			
VZEROUPPER	Zero All YMM Registers Upper	3		AVX			
WAIT	Wait for x87 Floating-Point Exceptions	3				X87	
WBINVD	Writeback and Invalidate Caches	0					Base
WRFSBASE	Write FS.base	3					FSGSBASE
WRGSBASE	Write GS.base	3					FSGSBASE
WRMSR	Write to Model-Specific Register	0					MSR
XADD	Exchange and Add	3	Base				
XCHG	Exchange	3	Base				
XGETBV	Get Extended Control Register Value	3		XSAVE			
XLAT	Translate Table Index	3	Base				
XLATB	Translate Table Index (No Operands)	3	Base				
XOR	Exclusive OR	3	Base				
XORPD	Logical Bitwise Exclusive OR Packed Double-Precision Floating-Point	3		SSE2			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. "Base" indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

**Table D-2. Instruction Groups and CPUID Feature Flags**

Instruction			Instruction Group and CPUID Feature Flag(s) <sup>1</sup>				
Mnemonic	Description	CPL	General-Purpose	SSE	64-Bit Media	x87	System
XORPS	Logical Bitwise Exclusive OR Packed Single-Precision Floating-Point	3		SSE			
XRSTOR	Restore Extended States	3		XSAVE			
XSAVE	Save Extended States	3		XSAVE			
XSAVEOPT	Save Extended States Performance Optimized	3		XSAVEOPT			
XSETBV	Set Extended Control Register Value	3		XSAVE			

**Notes:**

1. Columns indicate the instruction groups. Entries indicate the CPUID feature flag(s) indicating support for that instruction. “Base” indicates that no feature flag exists and all processors support this instruction.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.
3. MONITOR/MWAIT can execute at privilege level >0 if enabled. See instruction description.
4. One or more features of the instruction (usually support for 256-bit operands) are only available if AVX2 is supported.

# Appendix E Obtaining Processor Information Via the CPUID Instruction

This appendix specifies the information that software can obtain about the processor on which it is running by executing the CPUID instruction. The information in this appendix supersedes the contents of the *CPUID Specification*, order #25481, which is now obsolete.

The CPUID instruction is described on page 158. This appendix does not replace the CPUID instruction reference information presented there.

The CPUID instruction behaves much like a function call. Parameters are passed to the instruction via registers and on execution the instruction loads specific registers with return values. These return values can be interpreted by software based on the field definitions and their assigned meanings.

The first input parameter is the *function number* which is passed to the instruction via the EAX register. Some functions also accept a second input parameter passed via the ECX register. Values are returned via the EAX, EBX, ECX, and EDX registers. Software should not assume that any values written to these registers prior to the execution of CPUID instruction will be retained after the instruction executes (even those that are marked reserved).

The description of each return value breaks the value down into one or more named *fields* which represent a bit position or contiguous range of bits. All bit positions that are not defined as fields are reserved. The value of bits within reserved ranges cannot be relied upon to be zero. Software must mask off all reserved bits in the return value prior to making any value comparisons of represented information.

This appendix applies to all AMD processors with a family designation of 0Fh or greater.

## E.1 Special Notational Conventions

The following special notation conventions are used in this appendix:

- The notation (standard throughout this APM) for representing the function number, optional input parameter, and the information returned is as follows:

CPUID FnXXXX\_XXXX\_RRR[*FieldName*]\_xYYY.

Where:

- *XXXX\_XXXX* is the function number represented in hexadecimal (passed to the instruction in EAX).
- *RRR* is one of {EDX, ECX, EBX, EAX} and represents a register holding a return value.
- *YYY* represents the optional input parameter passed in the ECX register expressed as a hexadecimal number. If this parameter is not used, the characters represented by *\_xYYY* are omitted from the notation.

- *FieldName* identifies a specific named element of processor information represented by a specific bit range (1 or more bits wide) within the *RRR* register.
- The notation CPUID FnXXXX\_XXXX\_*RRR* is used when referring to one of the registers that holds information returned by the instruction.
- The notation CPUID FnXXXX\_XXXX or FnXXXX\_XXXX is used to refer to a specific function number.
- Most one-bit fields indicate support or non-support of a specific processor feature. By convention, (unless otherwise noted) a value of 1 means that the feature is supported by the processor and a value of 0 means that the feature is not supported by the processor.

## E.2 Standard and Extended Function Numbers

The CPUID instruction supports two sets or ranges of function numbers: standard and extended.

- The smallest function number of the standard function range is Fn0000\_0000. The largest function number of the standard function range, for a particular implementation, is returned in CPUID Fn0000\_0000\_EAX.
- The smallest function number of the extended function range is Fn8000\_0000. The largest function number of the extended function range, for a particular implementation, is returned in CPUID Fn8000\_0000\_EAX.

## E.3 Standard Feature Function Numbers

This section describes each of the defined CPUID functions in the standard range.

### E.3.1 Function 0h—Maximum Standard Function Number and Vendor String

This function number provides information about the maximum standard function number supported on this processor and a string that identifies the vendor of the product.

#### CPUID Fn0000\_0000\_EAX Largest Standard Function Number

The value returned in EAX provides the largest standard function number supported by this processor.

Bits	Field Name	Description
31:0	LFuncStd	Largest standard function. The largest CPUID standard function input value supported by the processor implementation.

#### CPUID Fn0000\_0000\_E[D,C,B]X Processor Vendor

The values returned in EBX, EDX, and ECX together provide a 12-character string identifying the vendor of this processor. Each register supplies 4 characters. The leftmost character of each substring

is stored in the least significant bit position in the register. The string is the concatenation of the contents of EBX, EDX, and ECX in left to right order. No null terminator is included in the string.

[CPUID Fn8000\\_0000\\_E\[D,C,B\]X](#) return the same values as this function.

Bits	Field Name	Description
31:0	Vendor	Four characters of the 12-byte character string (encoded in ASCII) “AuthenticAMD”. See Table E-1 below.

**Table E-1. CPUID Fn0000\_0000\_E[D,C,B]X values**

Register	Value	Description
CPUID Fn0000_0000_EBX	6874_7541h	The ASCII characters “h t u A”.
CPUID Fn0000_0000_ECX	444D_4163h	The ASCII characters “D M A c”.
CPUID Fn0000_0000_EDX	6974_6E65h	The ASCII characters “i t n e”.

### E.3.2 Function 1h—Processor and Processor Feature Identifiers

This function number identifies the processor family, model, and stepping and provides feature support information.

#### CPUID Fn0000\_0001\_EAX Family, Model, Stepping Identifiers

The value returned in EAX provides the family, model, and stepping identifiers. Three values are used by software to identify a processor: Family, Model, and Stepping.

Bits	Field Name	Description
31:28	—	Reserved.
27:20	ExtFamily	Processor extended family. See above for definition of Family[7:0].
19:16	ExtModel	Processor extended model. See above for definition of Model[7:0].
15:12	—	Reserved.
11:8	BaseFamily	Base processor family. See above for definition of Family[7:0].
7:4	BaseModel	Base processor model. See above for definition of Model[7:0].
3:0	Stepping	Processor stepping. Processor stepping (revision) for a specific model.

The processor *Family* identifies one or more processors as belonging to a group that possesses some common definition for software or hardware purposes. The *Model* specifies one instance of a processor family. The *Stepping* identifies a particular version of a specific model. Therefore, Family, Model and Stepping, when taken together, form a unique identification or signature for a processor.

The **Family** is an 8-bit value and is defined as: **Family[7:0]** = ({0000b,BaseFamily[3:0]} + ExtFamily[7:0]). For example, if BaseFamily[3:0] = Fh and ExtFamily[7:0] = 01h, then Family[7:0] =

10h. If BaseFamily[3:0] is less than Fh, then ExtFamily is reserved and Family is equal to BaseFamily[3:0].

**Model** is an 8-bit value and is defined as: **Model[7:0]** = {ExtModel[3:0],BaseModel[3:0]}. For example, if ExtModel[3:0] = Eh and BaseModel[3:0] = 8h, then Model[7:0] = E8h. If BaseFamily[3:0] is less than 0Fh, then ExtModel is reserved and Model is equal to BaseModel[3:0].

The value returned by [CPUID Fn8000\\_0001\\_EAX](#) is equivalent to [CPUID Fn0000\\_0001\\_EAX](#).

### **CPUID Fn0000\_0001\_EBX LocalApicId, LogicalProcessorCount, CLFlush**

---

The value returned in EBX provides miscellaneous information regarding the processor brand, the number of logical threads per processor socket, the CLFLUSH instruction, and APIC.

Bits	Field Name	Description
31:24	LocalApicId	Initial local APIC physical ID. The 8-bit value assigned to the local APIC physical ID register at power-up. Some of the bits of LocalApicId represent the core within a processor and other bits represent the processor ID. See the APIC20 “APIC ID” register in the processor BKDG for details.
23:16	LogicalProcessorCount	Logical processor count. If <a href="#">CPUID Fn0000_0001_EDX[HTT]</a> = 1 then LogicalProcessorCount is the number of cores per processor. If <a href="#">CPUID Fn0000_0001_EDX[HTT]</a> = 0 then LogicalProcessorCount is reserved. See <a href="#">E.5.1 [Legacy Method]</a> .
15:8	CLFlush	CLFLUSH size. Specifies the size of a cache line in quadwords flushed by the CLFLUSH instruction. See “CLFLUSH” in APM3.
7:0	8BitBrandId	8-bit brand ID. This field, in conjunction with <a href="#">CPUID Fn8000_0001_EBX[BrandId]</a> , is used by the system firmware to generate the processor name string. See the appropriate processor revision guide for how to program the processor name string.

### **CPUID Fn0000\_0001\_ECX Feature Identifiers**

---

The value returned in ECX contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31	—	RAZ. Reserved for use by hypervisor to indicate guest status.
30	RDRAND	RDRAND instruction support.
29	F16C	Half-precision convert instruction support. See “Half-Precision Floating-Point Conversion” in APM1 and listings for individual F16C instructions in APM5.
28	AVX	AVX instruction support. See APM4.
27	OSXSAVE	XSAVE (and related) instructions are enabled. See “OSXSAVE” in APM2..
26	XSAVE	XSAVE (and related) instructions are supported by hardware. See “XSAVE/XRSTOR Instructions” in APM2.

Bits	Field Name	Description
25	AES	AES instruction support. See “AES Instructions” in APM4.
24	—	Reserved.
23	POPCNT	POPCNT instruction. See “POPCNT” in APM3.
22		MOVBE: MOVBE instruction support.
21	—	Reserved.
20	SSE42	SSE4.2 instruction support. “Determining Media and x87 Feature Support” in APM2 and individual SSE4.2 instruction listings in APM4.
19	SSE41	SSE4.1 instruction support. See individual instruction listings in APM4. .
18:14	—	Reserved.
13	CMPXCHG16B	CMPXCHG16B instruction support. See “CMPXCHG16B” in APM3.
12	FMA	FMA instruction support.
11:10	—	Reserved.
9	SSSE3	Supplemental SSE3 instruction support.
8:4	—	Reserved.
3	MONITOR	MONITOR/MWAIT instructions. See “MONITOR” and “MWAIT” in APM3.
2	—	Reserved.
1	PCLMULQDQ	PCLMULQDQ instruction support. See instruction reference page for the PCLMULQDQ / VPCLMULQDQ instruction in APM4.
0	SSE3	SSE3 instruction support. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3 for the list of instructions covered by the SSE3 feature bit. See APM4 for the definition of the SSE3 instructions.

### CPUID Fn0000\_0001\_EDX Feature Identifiers

The value returned in EDX contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31:29	—	Reserved.
28	HTT	Hyper-threading technology. Indicates either that there is more than one thread per core or more than one core per processor. See “Legacy Method” on page 633.
27	—	Reserved.
26	SSE2	SSE2 instruction support. See Appendix D “CPUID Feature Sets” in APM3.
25	SSE	SSE instruction support. See Appendix D “CPUID Feature Sets” in APM3 appendix and “64-Bit Media Programming” in APM1.
24	FXSR	FXSAVE and FXRSTOR instructions. See “FXSAVE” and “FXRSTOR” in APM5.
23	MMX	MMX™ instructions. See Appendix D “CPUID Feature Sets” in APM3 and “128-Bit Media and Scientific Programming” in APM1.
22:20	—	Reserved.
19	CLFSH	CLFLUSH instruction support. See “CLFLUSH” in APM3.

Bits	Field Name	Description
18	—	Reserved.
17	PSE36	Page-size extensions. The PDE[20:13] supplies physical address [39:32]. See “Page Translation and Protection” in APM2.
16	PAT	Page attribute table. See “Page-Attribute Table Mechanism” in APM2.
15	CMOV	Conditional move instructions. See “CMOV”, “FCMOV” in APM3.
14	MCA	Machine check architecture. See “Machine Check Mechanism” in APM2.
13	PGE	Page global extension. See “Page Translation and Protection” in APM2.
12	MTRR	Memory-type range registers. See “Page Translation and Protection” in APM2.
11	SysEnterSysExit	SYSENTER and SYSEXIT instructions. See “SYSENTER”, “SYSEXIT” in APM3.
10	—	Reserved.
9	APIC	Advanced programmable interrupt controller. Indicates APIC exists and is enabled. See “Exceptions and Interrupts” in APM2.
8	CMPXCHG8B	CMPXCHG8B instruction. See “CMPXCHG8B” in APM3.
7	MCE	Machine check exception. See “Machine Check Mechanism” in APM2.
6	PAE	Physical-address extensions. Indicates support for physical addresses <sup>3</sup> 32b. Number of physical address bits above 32b is implementation specific. See “Page Translation and Protection” in APM2.
5	MSR	AMD model-specific registers. Indicates support for AMD model-specific registers (MSRs), with RDMSR and WRMSR instructions. See “Model Specific Registers” in APM2.
4	TSC	Time stamp counter. RDTSC and RDTSCP instruction support. See “Debug and Performance Resources” in APM2.
3	PSE	Page-size extensions. See “Page Translation and Protection” in APM2.
2	DE	Debugging extensions. See “Debug and Performance Resources” in APM2.
1	VME	Virtual-mode enhancements. CR4.VME, CR4.PVI, software interrupt indirection, expansion of the TSS with the software, indirection bitmap, EFLAGS.VIF, EFLAGS.VIP. See “System Resources” in APM2.
0	FPU	x87 floating point unit on-chip. See “x87 Floating Point Programming” in APM1.

### E.3.3 Functions 2h–4h—Reserved

#### CPUID Fn0000\_000[4:2] Reserved

These function numbers are reserved.

### E.3.4 Function 5h—Monitor and MWAIT Features

This function provides feature identifiers for the MONITOR and MWAIT instructions. For more information see the description of the MONITOR instruction on page 388 and the MWAIT instruction on page 394.

---

**CPUID Fn0000\_0005\_EAX Monitor/MWait**

---

The value returned in EAX provides the following information:

Bits	Field Name	Description
31:16	—	Reserved.
15:0	MonLineSizeMin	Smallest monitor-line size in bytes.

---

**CPUID Fn0000\_0005\_EBX Monitor/MWait**

---

The value returned in EBX provides the following information:

Bits	Field Name	Description
31:16	—	Reserved.
15:0	MonLineSizeMax	Largest monitor-line size in bytes.

---

**CPUID Fn0000\_0005\_ECX Monitor/MWait**

---

The value returned in ECX provides the following information:

Bits	Field Name	Description
31:2	—	Reserved.
1	IBE	Interrupt break-event. Indicates MWAIT can use ECX bit 0 to allow interrupts to cause an exit from the monitor event pending state, even if EFLAGS.IF=0.
0	EMX	Enumerate MONITOR/MWAIT extensions: Indicates enumeration MONITOR/MWAIT extensions are supported.

---

**CPUID Fn0000\_0005\_EDX Monitor/MWait**

---

The value returned in EDX is undefined and is reserved.

**E.3.5 Function 6h—Power Management Related Features**

This function provides information about the local APIC timer timebase and the effective frequency interface for the processor.

---

**CPUID Fn0000\_0006\_EAX Local APIC Timer Invariance**

---

The value returned in EAX is undefined and is reserved.

Bits	Field Name	Description
31:3	—	Reserved.
2	ARAT	If set, indicates that the timebase for the local APIC timer is not affected by processor p-state.
1:0	—	Reserved.

**CPUID Fn0000\_0006\_EBX Reserved**

The value returned in EBX is undefined and is reserved.

**CPUID Fn0000\_0006\_ECX Effective Processor Frequency Interface**

The value returned in ECX indicates support of the processor effective frequency interface. For more information on this feature, see "Determining Processor Effective Frequency" in APM2.

Bits	Field Name	Description
31:1	—	Reserved.
0	EffFreq	Effective frequency interface support. If set, indicates presence of MSR0000_00E7 (MPERF) and MSR0000_00E8 (APERF).

**CPUID Fn0000\_0006\_EDX Reserved**

The value returned in EDX is undefined and is reserved.

**E.3.6 Function 7h—Structured Extended Feature Identifiers****CPUID Fn0000\_0007\_EAX\_x0 Structured Extended Feature Identifiers (ECX=0)**

Bits	Field Name	Description
31:0	MaxSubFn	Returns the number of subfunctions supported.

**CPUID Fn0000\_0007\_EBX\_x0 Structured Extended Feature Identifiers (ECX=0)**

Bits	Field Name	Description
31:9	—	Reserved.
8	BMI2	Bit manipulation group 2 instruction support.
7	SMEP	Supervisor mode execution protection.
6	—	Reserved.
5	AVX2	AVX2 instruction subset support.
4	—	Reserved.
3	BMI1	Bit manipulation group 1 instruction support.

Bits	Field Name	Description
2:1	—	Reserved.
0	FSGSBASE	FS and GS base read write instruction support.

**CPUID Fn0000\_0007\_ECX\_x0 Structured Extended Feature Identifiers (ECX=0)**

Bits	Field Name	Description
31:0	—	Reserved.

**CPUID Fn0000\_0007\_EDX\_x0 Structured Extended Feature Identifiers (ECX=0)**

Bits	Field Name	Description
31:0	—	Reserved.

**E.3.7 Functions 8h–Ch—Reserved****CPUID Fn0000\_000[C:8] Reserved**

These function numbers are reserved.

**E.3.8 Function Dh—Processor Extended State Enumeration**

The XSAVE / XRSTOR instructions are used to save and restore x87/MMX FPU and SSE processor state. These instructions allow processor state associated with specific architected features to be selectively saved and restored. This function provides information about extended state support and save area size requirements.

The function has a number of subfunctions specified by the input value passed to the CPUID instruction in the ECX register.

**Subfunction 0 of Fn0000\_000D**

Subfunction 0 provides information about features within the extended processor state management architecture that are supported by the processor.

**CPUID Fn0000\_000D\_EAX\_x0 Processor Extended State Enumeration (ECX=0)**

The value returned in EAX provides a bit mask specifying which of the features defined by the extended processor state architecture are supported by the processor.

Bits	Field Name	Description
31:0	XFeatureSupportedMask[31:0]	Reports the valid bit positions for the lower 32 bits of the XFeatureEnabledMask register. If a bit is set, the corresponding feature is supported. See “XSAVE/XRSTOR Instructions” in APM2.

**CPUID Fn0000\_000D\_EBX\_x0 Processor Extended State Enumeration (ECX=0)**

The value returned in EBX gives the save area size requirement in bytes based on the features currently enabled in the XFEATURE\_ENABLED\_MASK (XCR0).

Bits	Field Name	Description
31:0	XFeatureEnabledSizeMax	Size in bytes of XSAVE/XRSTOR area for the currently enabled features in XCR0.

**CPUID Fn0000\_000D\_ECX\_x0 Processor Extended State Enumeration (ECX=0)**

The value returned in ECX gives the save area size requirement in bytes for all extended state management features supported by the processor (whether enabled or not).

Bits	Field Name	Description
31:0	XFeatureSupportedSizeMax	Size in bytes of XSAVE/XRSTOR area for all features that the core supports. See XFeatureEnabledSizeMax.

**CPUID Fn0000\_000D\_EDX\_x0 Processor Extended State Enumeration (ECX=0)**

The value returned in EDX provides a bit mask specifying which of the features defined by the extended processor state architecture are supported by the processor.

Bits	Field Name	Description
31:0	XFeatureSupportedMask[63:32]	Reports the valid bit positions for the upper 32 bits of the XFeatureEnabledMask register. If a bit is set, the corresponding feature is supported.

See “XSAVE/XRSTOR Instructions” in APM2 and reference pages for the individual instructions in APM4.

**Subfunction 1 of Fn0000\_000D**

Subfunction 1 provides additional information about features within the extended processor state management architecture that are supported by the processor.

**CPUID Fn0000\_000D\_EAX\_x1 Processor Extended State Enumeration (ECX=1)**

Bits	Field Name	Description
31:1		Reserved.
0	XSAVEOPT	XSAVEOPT is available.

**CPUID Fn0000\_000D\_E[D,C,B]X\_x1 Processor Extended State Enumeration (ECX=1)**

The values returned in EBX, ECX, and EDX for subfunction 1 are undefined and are reserved.

### **Subfunction 2 of Fn0000\_000D**

Subfunction 2 provides information about the size and offset of the 256-bit SSE vector floating point processor unit state save area.

#### **CPUID Fn0000\_000D\_EAX\_x2 Processor Extended State Enumeration (ECX=2)**

The value returned in EAX provides information about the size of the 256-bit SSE vector floating point processor unit state save area.

Bits	Field Name	Description
31:0	YmmSaveStateSize	YMM state save size. The state save area size in bytes for The YMM registers.

#### **CPUID Fn0000\_000D\_EBX\_x2 Processor Extended State Enumeration (ECX=2)**

The value returned in EBX provides information about the offset of the 256-bit SSE vector floating point processor unit state save area from the base of the extended state (XSAVE/XRSTOR) save area.

Bits	Field Name	Description
31:0	YmmSaveStateOffset	YMM state save offset. The offset in bytes from the base of the extended state save area of the YMM register state save area.

#### **CPUID Fn0000\_000D\_E[D,C]X\_x2 Processor Extended State Enumeration (ECX=2)**

The values returned in ECX and EDX for subfunction 2 are undefined and are reserved.

If CPUID Fn0000\_000D is executed with a subfunction (passed in ECX) greater than 2 but less than 3Eh, the instruction returns all zeros in the EAX, EBX, ECX, and EDX registers.

### **Subfunction 3Eh of Fn0000\_000D**

Subfunction 3Eh provides information about the size and offset of the Lightweight Profiling (LWP) unit state save area.

#### **CPUID Fn0000\_000D\_EAX\_x3E Processor Extended State Enumeration (ECX=62)**

The value returned in EAX provides the size of the Lightweight Profiling (LWP) unit state save area.

Bits	Field Name	Description
31:0	LwpSaveStateSize	LWP state save area size. The size of the save area for LWP state in bytes. See “Lightweight Profiling” in APM2.

**CPUID Fn0000\_000D\_EBX\_x3E Processor Extended State Enumeration (ECX=62)**

The value returned in EBX provides the offset of the Lightweight Profiling (LWP) unit state save area from the base of the extended state (XSAVE/XRSTOR) save area.

Bits	Field Name	Description
31:0	LwpSaveStateOffset	LWP state save byte offset. The offset in bytes from the base of the extended state save area of the state save area for LWP. See “Lightweight Profiling” in APM2.

**CPUID Fn0000\_000D\_E[D,C]X\_x3E Processor Extended State Enumeration (ECX=62)**

The values returned in ECX and EDX for subfunction 3Eh are undefined and are reserved.

**Subfunctions of Fn0000\_000D greater than 3Eh**

For CPUID Fn0000\_000D, if the subfunction (specified by contents of ECX) passed as input to the instruction is greater than 3Eh, the instruction returns zero in the EAX, EBX, ECX, and EDX registers.

**E.3.9 Functions 4000\_0000h–4000\_FFh—Reserved for Hypervisor Use****CPUID Fn4000\_00[FF:00] Reserved**

These function numbers are reserved for use by the virtual machine monitor.

**E.4 Extended Feature Function Numbers**

This section describes each of the defined CPUID functions in the extended range.

**E.4.1 Function 8000\_0000h—Maximum Extended Function Number and Vendor String**

This function provides information about the maximum extended function number supported on this processor and a string that identifies the vendor of the product.

**CPUID Fn8000\_0000\_EAX Largest Extended Function Number**

The value returned in EAX provides the largest extended function number supported by the processor.

Bits	Field Name	Description
31:0	LFuncExt	Largest extended function. The largest CPUID extended function input value supported by the processor implementation.

**CPUID Fn8000\_0000\_E[D,C,B]X Processor Vendor**

The values returned in EBX, ECX, and EDX together provide a 12-character string identifying the vendor of this processor. The output string is the same as the one returned by Fn0000\_0000. See [CPUID Fn0000\\_0000\\_E\[D,C,B\]X](#) on page 602 for more details.

Bits	Field Name	Description
31:0	Vendor	Four characters of the 12-byte character string (encoded in ASCII) “AuthenticAMD”. See Table E-2 below.

**Table E-2. CPUID Fn8000\_0000\_E[D,C,B]X values**

Register	Value	Description
CPUID Fn8000_0000_EBX	6874_7541h	The ASCII characters “h t u A”.
CPUID Fn8000_0000_ECX	444D_4163h	The ASCII characters “D M A c”.
CPUID Fn8000_0000_EDX	6974_6E65h	The ASCII characters “i t n e”.

**E.4.2 Function 8000\_0001h—Extended Processor and Processor Feature Identifiers****CPUID Fn8000\_0001\_EAX AMD Family, Model, Stepping**

The value returned in EAX provides the family, model, and stepping identifiers. Three values are used by software to identify a processor: Family, Model, and Stepping. The value returned in EAX is the same as the value returned in EAX for Fn0000\_0001. See [CPUID Fn0000\\_0001\\_EAX](#) on page 603 for more details on the field definitions.

Bits	Field Names	Description
31:0	Family, Model, Stepping	See: <a href="#">CPUID Fn0000_0001_EAX</a> .

**CPUID Fn8000\_0001\_EBX BrandId Identifier**

The value returned in EBX provides package type and a 16-bit processor name string identifiers.

Bits	Field Name	Description
31:28	PkgType	Package type. If (Family[7:0] >= 10h), this field is valid. If (Family[7:0]<10h), this field is reserved.
27:16	—	Reserved.
15:0	BrandId	Brand ID. This field, in conjunction with <a href="#">CPUID Fn0000_0001_EBX[8BitBrandId]</a> , is used by system firmware to generate the processor name string. See your processor revision guide for how to program the processor name string.

For processor families 10h and greater, PkgType is described in the *BIOS and Kernel Developer’s Guide* for the product.

### **CPUID Fn8000\_0001\_ECX Feature Identifiers**

---

This function contains the following miscellaneous feature identifiers:

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:28	—	Reserved.
27	PerfTsc	Performance time-stamp counter. Indicates support for MSRC001_0280 [Performance Time Stamp Counter].
26	DataBreakpointEx tension	Data access breakpoint extension. Indicates support for MSRC001_1027 and MSRC001_101[B:9].
25	—	Reserved
24	PerfCtrExtNB	NB performance counter extensions support. Indicates support for MSRC001_024[6,4,2,0] and MSRC001_024[7,5,3,1].
23	PerfCtrExtCore	Processor performance counter extensions support. Indicates support for MSRC001_020[A,8,6,4,2,0] and MSRC001_020[B,9,7,5,3,1].
22	TopologyExtensions	Topology extensions support. Indicates support for <a href="#">CPUID Fn8000_001D_EAX_x[N:0]-CPUID Fn8000_001E_EDX</a> .
21	TBM	Trailing bit manipulation instruction support.
20	—	Reserved.
19	—	Reserved.
18	—	Reserved.
17	—	Reserved.
16	FMA4	Four-operand FMA instruction support.
15	LWP	Lightweight profiling support. See “Lightweight Profiling” in APM2 and reference pages for individual LWP instructions in APM3.
14	—	Reserved.
13	WDT	Watchdog timer support. See APM2 and APM3. Indicates support for MSRC001_0074.
12	SKINIT	SKINIT and STGI are supported. Indicates support for SKINIT and STGI, independent of the value of MSRC000_0080[SVME]. See APM2 and APM3.
11	XOP	Extended operation support.
10	IBS	Instruction based sampling. See “Instruction Based Sampling” in APM2.
9	OSVW	OS visible workaround. Indicates OS-visible workaround support. See “OS Visible Work-around (OSVW) Information” in APM2.
8	3DNowPrefetch	PREFETCH and PREFETCHW instruction support. See “PREFETCH” and “PREFETCHW” in APM3.
7	MisAlignSse	Misaligned SSE mode. See “Misaligned Access Support Added for SSE Instructions” in APM1.
6	SSE4A	EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD instruction support. See “EXTRQ”, “INSERTQ”, “MOVNTSS”, and “MOVNTSD” in APM4.

Bits	Field Name	Description
5	ABM	Advanced bit manipulation. LZCNT instruction support. See “LZCNT” in APM3.
4	AltMovCr8	LOCK MOV CR0 means MOV CR8. See “MOV(CRn)” in APM3.
3	ExtApicSpace	Extended APIC space. This bit indicates the presence of extended APIC register space starting at offset 400h from the “APIC Base Address Register,” as specified in the BKDG.
2	SVM	Secure virtual machine. See “Secure Virtual Machine” in APM2.
1	CmpLegacy	Core multi-processing legacy mode. See “Legacy Method” on page 633.
0	LahfSahf	LAHF and SAHF instruction support in 64-bit mode. See “LAHF” and “SAHF” in APM3.

### **CPUID Fn8000\_0001\_EDX Feature Identifiers**

This function contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31	3DNow	3DNow!™ instructions. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3.
30	3DNowExt	AMD extensions to 3DNow! instructions. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3.
29	LM	Long mode. See “Processor Initialization and Long-Mode Activation” in APM2.
28	—	Reserved.
27	RDTSCP	RDTSCP instruction. See “RDTSCP” in APM3.
26	Page1GB	1-GB large page support. See “1-GB Paging Support” in APM2.
25	FFXSR	FXSAVE and FXRSTOR instruction optimizations. See “FXSAVE” and “FXRSTOR” in APM5.
24	FXSR	FXSAVE and FXRSTOR instructions. Same as <a href="#">CPUID Fn0000_0001_EDX[FXSR]</a> .
23	MMX	MMX™ instructions. Same as <a href="#">CPUID Fn0000_0001_EDX[MMX]</a> .
22	MmxExt	AMD extensions to MMX instructions. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3 and “128-Bit Media and Scientific Programming” in APM1.
21	—	Reserved.
20	NX	No-execute page protection. See “Page Translation and Protection” in APM2.
19:18	—	Reserved.
17	PSE36	Page-size extensions. Same as <a href="#">CPUID Fn0000_0001_EDX[PSE36]</a> .
16	PAT	Page attribute table. Same as <a href="#">CPUID Fn0000_0001_EDX[PAT]</a> .
15	CMOV	Conditional move instructions. Same as <a href="#">CPUID Fn0000_0001_EDX[CMOV]</a> .
14	MCA	Machine check architecture. Same as <a href="#">CPUID Fn0000_0001_EDX[MCA]</a> .
13	PGE	Page global extension. Same as <a href="#">CPUID Fn0000_0001_EDX[PGE]</a> .
12	MTRR	Memory-type range registers. Same as <a href="#">CPUID Fn0000_0001_EDX[MTRR]</a> .
11	SysCallSysRet	SYSCALL and SYSRET instructions. See “SYSCALL” and “SYSRET” in APM3.

Bits	Field Name	Description
10	—	Reserved.
9	APIC	Advanced programmable interrupt controller. Same as <a href="#">CPUID Fn0000_0001_EDX[APIC]</a> .
8	CMPXCHG8B	CMPXCHG8B instruction. Same as <a href="#">CPUID Fn0000_0001_EDX[CMPXCHG8B]</a> .
7	MCE	Machine check exception. Same as <a href="#">CPUID Fn0000_0001_EDX[MCE]</a> .
6	PAE	Physical-address extensions. Same as <a href="#">CPUID Fn0000_0001_EDX[PAE]</a> .
5	MSR	AMD model-specific registers. Same as <a href="#">CPUID Fn0000_0001_EDX[MSR]</a> .
4	TSC	Time stamp counter. Same as <a href="#">CPUID Fn0000_0001_EDX[TSC]</a> .
3	PSE	Page-size extensions. Same as <a href="#">CPUID Fn0000_0001_EDX[PSE]</a> .
2	DE	Debugging extensions. Same as <a href="#">CPUID Fn0000_0001_EDX[DE]</a> .
1	VME	Virtual-mode enhancements. Same as <a href="#">CPUID Fn0000_0001_EDX[VME]</a> .
0	FPU	x87 floating-point unit on-chip. Same as <a href="#">CPUID Fn0000_0001_EDX[FPU]</a> .

#### E.4.3 Functions 8000\_0002h–8000\_0004h—Extended Processor Name String

##### **CPUID Fn8000\_000[4:2]\_E[D,C,B,A]X Processor Name String Identifier**

The three extended functions from Fn8000\_0002 to Fn8000\_0004 are programmed to return a null terminated ASCII string up to 48 characters in length corresponding to the processor name.

Bits	Field Name	Description
31:0	ProcName	Four characters of the extended processor name string.

The 48 character maximum includes the terminating null character. The 48 character string is ordered first to last (left to right) as follows:

Fn8000\_0002[EAX[7:0],..., EAX[31:24], EBX[7:0],..., EBX[31:24], ECX[7:0],..., ECX[31:24],EDX[7:0],..., EDX[31:24]],

Fn8000\_0003[EAX[7:0],..., EAX[31:24], EBX[7:0],..., EBX[31:24], ECX[7:0],..., ECX[31:24], EDX[7:0],..., EDX[31:24]],

Fn8000\_0004[EAX[7:0],..., EAX[31:24], EBX[7:0],..., EBX[31:24], ECX[7:0],..., ECX[31:24], EDX[7:0],..., EDX[31:24]].

The extended processor name string is programmed by system firmware. See your processor revision guide for information about how to display the extended processor name string.

#### E.4.4 Function 8000\_0005h—L1 Cache and TLB Information

This function provides first level cache TLB characteristics for the processor that executes the instruction.

##### **CPUID Fn8000\_0005\_EAX L1 TLB 2M/4M Information**

The value returned in EAX provides information about the L1 TLB for 2-MB and 4-MB pages.

Bits	Field Name	Description
31:24	L1DTlb2and4MAssoc	Data TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-3 below.
23:16	L1DTlb2and4MSize	Data TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2-MB page size; 4-MB pages require two 2-MB entries, so the number of entries available for the 4-MB page size is one-half the returned value.
15:8	L1ITlb2and4MAssoc	Instruction TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-3 below.
7:0	L1ITlb2and4MSize	Instruction TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2-MB page size; 4-MB pages require two 2-MB entries, so the number of entries available for the 4-MB page size is one-half the returned value.

The associativity fields (L1DTlb2and4MAssoc and L1ITlb2and4MAssoc) are encoded as follows:

**Table E-3. L1 Cache and TLB Associativity Field Encodings**

Associativity [7:0]	Definition
00h	Reserved
01h	1 way (direct mapped)
02h–FEh	$n$ -way associative. (field encodes $n$ )
FFh	Fully associative

### CPUID Fn8000\_0005\_EBX L1 TLB 4K Information

The value returned in EBX provides information about the L1 TLB for 4-KB pages.

Bits	Field Name	Description
31:24	L1DTlb4KAssoc	Data TLB associativity for 4 KB pages. Encoding is per Table E-3 above.
23:16	L1DTlb4KSize	Data TLB number of entries for 4 KB pages.
15:8	L1ITlb4KAssoc	Instruction TLB associativity for 4 KB pages. Encoding is per Table E-3 above.
7:0	L1ITlb4KSize	Instruction TLB number of entries for 4 KB pages.

The associativity fields (L1DTlb4KAssoc and L1ITlb4KAssoc) are encoded as specified in Table E-3 on page 617.

### CPUID Fn8000\_0005\_ECX L1 Data Cache Information

The value returned in ECX provides information about the first level data cache.

Bits	Field Name	Description
31:24	L1DcSize	L1 data cache size in KB.
23:16	L1DcAssoc	L1 data cache associativity. Encoding is per Table E-3.
15:8	L1DcLinesPerTag	L1 data cache lines per tag.
7:0	L1DcLineSize	L1 data cache line size in bytes.

The associativity field (L1DcAssoc) is encoded as specified in Table E-3 on page 617.

#### **CPUID Fn8000\_0005\_EDX L1 Instruction Cache Information**

The value returned in EDX provides information about the first level instruction cache.

Bits	Field Name	Description
31:24	L1IcSize	L1 instruction cache size KB.
23:16	L1IcAssoc	L1 instruction cache associativity. Encoding is per Table E-3.
15:8	L1IcLinesPerTag	L1 instruction cache lines per tag.
7:0	L1IcLineSize	L1 instruction cache line size in bytes.

The associativity field (L1IcAssoc) is encoded as specified in Table E-3 on page 617.

#### **E.4.5 Function 8000\_0006h—L2 Cache and TLB and L3 Cache Information**

This function provides the second level cache and TLB characteristics for the processor that executes the instruction. The EDX register returns the processor’s third level cache characteristics that are shared by all cores of the processor.

#### **CPUID Fn8000\_0006\_EAX L2 TLB 2M/4M Information**

The value returned in EAX provides information about the L2 TLB for 2-MB and 4-MB pages.

Bits	Field Name	Description
31:28	L2DTlb2and4MAssoc	L2 data TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-4 below.
27:16	L2DTlb2and4MSize	L2 data TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2 MB page size; 4 MB pages require two 2 MB entries, so the number of entries available for the 4 MB page size is one-half the returned value.
15:12	L2ITlb2and4MAssoc	L2 instruction TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-4 below.
11:0	L2ITlb2and4MSize	L2 instruction TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2 MB page size; 4 MB pages require two 2 MB entries, so the number of entries available for the 4 MB page size is one-half the returned value.

The associativity fields (L2DTlb2and4MAssoc and L2ITlb2and4MAssoc) are encoded as follows:

**Table E-4. L2/L3 Cache and TLB Associativity Field Encoding**

Associativity [3:0]	Definition
0h	L2/L3 cache or TLB is disabled.
1h	Direct mapped.
2h	2-way associative.
3h	3-way associative.
4h	4-way associative.
5h	6-way associative.
6h	8-way associative.
8h	16-way associative.
9h	Value for all fields should be determined from Fn8000_001D
Ah	32-way associative.
Bh	48-way associative.
Ch	64-way associative.
Dh	96-way associative.
Eh	128-way associative.
Fh	Fully associative.
All other encodings are reserved.	

### CPUID Fn8000\_0006\_EBX L2 TLB 4K Information

The value returned in EBX provides information about the L2 TLB for 4-KB pages.

Bits	Field Name	Description
31:28	L2DTlb4KAssoc	L2 data TLB associativity for 4-KB pages. Encoding is per Table E-4 above.
27:16	L2DTlb4KSize	L2 data TLB number of entries for 4-KB pages.
15:12	L2ITlb4KAssoc	L2 instruction TLB associativity for 4-KB pages. Encoding is per Table E-4 above.
11:0	L2ITlb4KSize	L2 instruction TLB number of entries for 4-KB pages.

The associativity fields (L2DTlb4KAssoc and L2ITlb4KAssoc) are encoded per Table E-4 above.

### CPUID Fn8000\_0006(ECX L2 Cache Information

The value returned in ECX provides information about the L2 cache.

Bits	Field Name	Description
31:16	L2Size	L2 cache size in KB.
15:12	L2Assoc	L2 cache associativity. Encoding is per Table E-4 on page 619.
11:8	L2LinesPerTag	L2 cache lines per tag.
7:0	L2LineSize	L2 cache line size in bytes.

The associativity field (L2Assoc) is encoded per Table E-4 on page 619.

#### **CPUID Fn8000\_0006\_EDX L3 Cache Information**

The value returned in EDX provides the third level cache characteristics shared by all cores of a physical processor.

Bits	Field Name	Description
31:18	L3Size	Specifies the L3 cache size range: $(L3Size[31:18] * 512KB) \leq L3\ cache\ size < ((L3Size[31:18]+1) * 512KB)$ .
17:16	—	Reserved.
15:12	L3Assoc	L3 cache associativity. Encoded per Table E-4 on page 619.
11:8	L3LinesPerTag	L3 cache lines per tag.
7:0	L3LineSize	L3 cache line size in bytes.

The associativity field (L3Assoc) is encoded per Table E-4 on page 619.

#### **E.4.6 Function 8000\_0007h—Processor Power Management and RAS Capabilities**

This function provides information about the power management, power reporting, and RAS capabilities of the processor that executes the instruction. There may be other processor-specific features and reporting capabilities not covered here. Refer to the *BIOS and Kernel Developer’s Guide* for your specific product to obtain more information.

#### **CPUID Fn8000\_0007\_EAX Reserved**

Bits	Field Name	Description
31:0	—	Reserved.

#### **CPUID Fn8000\_0007\_EBX RAS Capabilities**

The value returned in EBX provides information about RAS features that allow system software to detect specific hardware errors.

Bits	Field Name	Description
31:3	—	Reserved.
2	HWA	Hardware assert supported. Indicates support for MSRC001_10[DF:C0].
1	SUCCOR	Software uncorrectable error containment and recovery capability. The processor supports software containment of uncorrectable errors through context synchronizing data poisoning and deferred error interrupts; see APM2, Chapter 9, “Determining Machine-Check Architecture Support.”
0	McaOverflowRecov	MCA overflow recovery support. If set, indicates that MCA overflow conditions (MCI_STATUS[Overflow]=1) are not fatal; software may safely ignore such conditions. If clear, MCA overflow conditions require software to shut down the system. See APM2, Chapter 9, “Handling Machine Check Exceptions.”

### **CPUID Fn8000\_0007\_ECX Processor Power Monitoring Interface**

The value returned in ECX provides information about the implementation of the processor power monitoring interface.

Bits	Field Name	Description
31:0	CpuPwrSampleTimeRatio	Specifies the ratio of the compute unit power accumulator sample period to the TSC counter period.

### **CPUID Fn8000\_0007\_EDX Advanced Power Management Features**

The value returned in EDX provides information about the advanced power management and power reporting features available. Refer to the *BIOS and Kernel Developer’s Guide* for your specific product for a detailed description of the definition of each power management feature.

Bits	Field Name	Description
31:13	—	Reserved.
12	ProcPowerReporting	Core power reporting interface supported.
11	ProcFeedbackInterface	Processor feedback interface. Value: 1. 1=Indicates support for processor feedback interface. <b>Note:</b> This feature is deprecated.
10	EffFreqRO	Read-only effective frequency interface. 1=Indicates presence of MSRC000_00E7 [Read-Only Max Performance Frequency Clock Count (MPerfReadOnly)] and MSRC000_00E8 [Read-Only Actual Performance Frequency Clock Count (APerfReadOnly)].
9	CPB	Core performance boost.
8	TscInvariant	TSC invariant. The TSC rate is ensured to be invariant across all P-States, C-States, and stop grant transitions (such as STPCLK Throttling); therefore the TSC is suitable for use as a source of time. 0 = No such guarantee is made and software should avoid attempting to use the TSC as a source of time.
7	HwPstate	Hardware P-state control. MSRC001_0061 [P-state Current Limit], MSRC001_0062 [P-state Control] and MSRC001_0063 [P-state Status] exist.

6	100MHzSteps	100 MHz multiplier Control.
5	—	Reserved.
4	TM	Hardware thermal control (HTC).
3	TTP	THERMTRIP.
2	VID	Voltage ID control. Function replaced by HwPstate.
1	FID	Frequency ID control. Function replaced by HwPstate.
0	TS	Temperature sensor.

#### E.4.7 Function 8000\_0008h—Processor Capacity Parameters and Extended Feature Identification

This function provides the size or capacity of various architectural parameters that vary by implementation, as well as an extension to the Fn8000\_0001 feature identifiers.

##### CPUID Fn8000\_0008\_EAX Long Mode Size Identifiers

The value returned in EAX provides information about the maximum host and guest physical and linear address width (in bits) supported by the processor.

Bits	Field Name	Description
31:24	—	Reserved.
23:16	GuestPhysAddrSize	Maximum guest physical byte address size in bits. This number applies only to guests using nested paging. When this field is zero, refer to the PhysAddrSize field for the maximum guest physical address size. See “Secure Virtual Machine” in APM2.
15:8	LinAddrSize	Maximum linear byte address size in bits.
7:0	PhysAddrSize	Maximum physical byte address size in bits. When GuestPhysAddrSize is zero, this field also indicates the maximum guest physical address size.

The address width reported is the maximum supported in any mode. For long mode capable processors, the size reported is independent of whether long mode is enabled. See “Processor Initialization and Long-Mode Activation” in APM2.

##### CPUID Fn8000\_0008\_EBX Extended Feature Identifiers

The value returned in EBX is an extension to the Fn8000\_0001 feature flags and indicates the presence of various ISA extensions.

Bit	Field Name	Description
0	CLZERO	CLZERO instruction supported
1	InstRetCntMsr	Instruction Retired Counter MSR available
2	RstrFpErrPtrs	FP Error Pointers Restored by XRSTOR

## **CPUID Fn8000\_0008\_ECX Size Identifiers**

The value returned in ECX provides information about the number of cores supported by the processor, the width of the APIC ID, and the width of the performance time-stamp counter.

Bits	Field Name	Description										
31:16	—	Reserved.										
17:16	PerfTscSize	<p>Performance time-stamp counter size. Indicates the size of MSRC001_0280[PTSC].</p> <table> <thead> <tr> <th>Bits</th><th>Description</th></tr> </thead> <tbody> <tr> <td>00b</td><td>40 bits</td></tr> <tr> <td>01b</td><td>48 bits</td></tr> <tr> <td>10b</td><td>56 bits</td></tr> <tr> <td>11b</td><td>64 bits</td></tr> </tbody> </table>	Bits	Description	00b	40 bits	01b	48 bits	10b	56 bits	11b	64 bits
Bits	Description											
00b	40 bits											
01b	48 bits											
10b	56 bits											
11b	64 bits											
15:12	ApicIdCoreIdSize	<p>APIC ID size. The number of bits in the initial APIC20[ApicId] value that indicate core ID within a processor. A zero value indicates that legacy methods must be used to derive the maximum number of cores. The size of this field determines the maximum number of cores (MNC) that the processor could theoretically support, not the actual number of cores that are actually implemented or enabled on the processor, as indicated by <a href="#">CPUID Fn8000_0008_ECX[NC]</a>.</p> <pre> if (ApicIdCoreIdSize[3:0] == 0){     // Used by legacy dual-core/single-core processors     MNC = CPUID Fn8000_0008_ECX[NC] + 1; } else {     // use ApicIdCoreIdSize[3:0] field     MNC = (2 ^ ApicIdCoreIdSize[3:0]); } </pre>										
11:8	—	Reserved.										
7:0	NC	Number of physical cores - 1. The number of cores in the processor is NC+1 (e.g., if NC = 0, then there is one core). See “Legacy Method” on page 633.										

## **CPUID Fn8000\_0008\_EDX Reserved**

The value returned in EDX for this function is undefined and is reserved.

### **E.4.8 Function 8000\_0009h—Reserved**

#### **CPUID Fn8000\_0009 Reserved**

This function is reserved.

### E.4.9 Function 8000\_000Ah—SVM Features

This function provides information about the SVM features that the processor supports. If SVM is not supported ([CPUID Fn8000\\_0001\\_ECX\[SVM\]](#) = 0), this function is reserved.

#### **CPUID Fn8000\_000A\_EAX SVM Revision and Feature Identification**

The value returned in EAX provides the SVM revision number. I

Bits	Field Name	Description
31:8	—	Reserved.
7:0	SvmRev	SVM revision number.

#### **CPUID Fn8000\_000A\_EBX SVM Revision and Feature Identification**

The value returned in EBX provides the number of address space identifiers (ASIDs) that the processor supports.

Bits	Field Name	Description
31:0	NASID	Number of available address space identifiers (ASID).

#### **CPUID Fn8000\_000A\_ECX Reserved**

The value returned in ECX for this function is undefined and is reserved.

#### **CPUID Fn8000\_000A\_EDX SVM Feature Identification**

The value returned in EDX provides Secure Virtual Machine architecture feature information. All cross references in the table below are to sections within the *Secure Virtual Machine* chapter of APM2.

Bits	Field Name	Description
31:17	—	Reserved.
16	VGIF	Virtualize the Global Interrupt Flag.. See "Nested Virtualization"
15	VMSAVEvirt	VMSAVE and VMLOAD virtualization. See "Nested Virtualization"
14	—	Reserved
13	AVIC	Support for the AMD advanced virtual interrupt controller. See "Advanced Virtual Interrupt Controller."
12	PauseFilterThreshold	PAUSE filter threshold. Indicates support for the PAUSE filter cycle count threshold. See "Pause Intercept Filtering."
11	—	Reserved.
10	PauseFilter	Pause intercept filter. Indicates support for the pause intercept filter. See "Pause Intercept Filtering."
9:8	—	Reserved.

Bits	Field Name	Description
7	DecodeAssists	Decode assists. Indicates support for the decode assists. See “Decode Assists.”
6	FlushByAsid	Flush by ASID. Indicates that TLB flush events, including CR3 writes and CR4.PGE toggles, flush only the current ASID’s TLB entries. Also indicates support for the extended VMCB TLB_Control. See “TLB Control.”
5	VmcbClean	VMCB clean bits. Indicates support for VMCB clean bits. See “VMCB Clean Bits.”
4	TscRateMsr	MSR based TSC rate control. Indicates support for MSR TSC ratio MSRC000_0104. See “TSC Ratio MSR (C000_0104h).”
3	NRIPS	NRIP save. Indicates support for NRIP save on #VMEXIT. See “State Saved on Exit.”
2	SVML	SVM lock. Indicates support for SVM-Lock. See “Enabling SVM.”
1	LbrVirt	LBR virtualization. Indicates support for LBR Virtualization. See “Enabling LBR Virtualization.”
0	NP	Nested paging. Indicates support for nested paging. See “Nested Paging.”

#### E.4.10 Functions 8000\_000Bh–8000\_0018h—Reserved

##### CPUID Fn8000\_00[18:0B] Reserved

These functions are reserved.

#### E.4.11 Function 8000\_0019h—TLB Characteristics for 1GB pages

This function provides information about the TLB for 1 GB pages for the processor that executes the instruction.

##### CPUID Fn8000\_0019\_EAX L1 TLB 1G Information

The value returned in EAX provides information about the L1 TLB for 1 GB pages.

Bits	Field Name	Description
31:28	L1DTlb1GAssoc	L1 data TLB associativity for 1 GB pages. See Table E-4 on page 619.
27:16	L1DTlb1GSize	L1 data TLB number of entries for 1 GB pages.
15:12	L1ITlb1GAssoc	L1 instruction TLB associativity for 1 GB pages. See Table E-4 on page 619.
11:0	L1ITlb1GSize	L1 instruction TLB number of entries for 1 GB pages.

##### CPUID Fn8000\_0019\_EBX L2 TLB 1G Information

The value returned in EBX provides information about the L2 TLB for 1 GB pages.

Bits	Field Name	Description
31:28	L2DTlb1GAssoc	L2 data TLB associativity for 1 GB pages. See Table E-4 on page 619.
27:16	L2DTlb1GSize	L2 data TLB number of entries for 1 GB pages.
15:12	L2ITlb1GAssoc	L2 instruction TLB associativity for 1 GB pages. See Table E-4 on page 619.
11:0	L2ITlb1GSize	L2 instruction TLB number of entries for 1 GB pages.

**CPUID Fn8000\_0019\_E[D,C]X Reserved**

The values returned in ECX and EDX for this function are undefined and reserved for future use.

**E.4.12 Function 8000\_001Ah—Instruction Optimizations****CPUID Fn8000\_001A\_EAX Performance Optimization Identifiers**

This function returns performance related information. For more details on how to use these bits to optimize software, see the *Software Optimization Guide* applicable to your product.

Bits	Field Name	Description
31:3	—	Reserved.
2	FP256	256-bit AVX instructions are executed with full-width internal operations and pipelines rather than decomposing them into internal 128-bit suboperations. This may impact how software performs instruction selection and scheduling.
1	MOVU	MOVU SSE instructions are more efficient and should be preferred to SSE MOVL/MOVH. MOVUPS is more efficient than MOVLPS/MOVHPS. MOVUPD is more efficient than MOVLPD/MOVHPD.
0	FP128	128-bit SSE (multimedia) instructions are executed with full-width internal operations and pipelines rather than decomposing them into internal 64-bit suboperations. This may impact how software performs instruction selection and scheduling.

**CPUID Fn8000\_001A\_E[D,C,B]X Reserved**

The values returned in EBX, ECX, and EDX are undefined for this function and are reserved.

#### E.4.13 Function 8000\_001Bh—Instruction-Based Sampling Capabilities

If instruction-based sampling (IBS) is supported ([CPUID Fn8000\\_0001\\_ECX](#)[IBS] = 1), this CPUID function can be used to obtain IBS feature information. If IBS is not supported ([CPUID Fn8000\\_0001\\_ECX](#)[IBS] = 0), this function number is reserved. For more information on using IBS, see “Instruction-Based Sampling” in APM2.

##### **CPUID Fn8000\_001B\_EAX Instruction-Based Sampling Feature Indicators**

The value returned in EAX provides the following information about the specific features of IBS that the processor supports:

Bits	Field Name	Description
31:9		Reserved.
8	OpBrnFuse	Fused branch micro-op indication supported.
7	RipInvalidChk	Invalid RIP indication supported.
6	OpCntExt	IbsOpCurCnt and IbsOpMaxCnt extend by 7 bits.
5	BrnTrgt	Branch target address reporting supported.
4	OpCnt	Op counting mode supported.
3	RdWrOpCnt	Read write of op counter supported.
2	OpSam	IBS execution sampling supported.
1	FetchSam	IBS fetch sampling supported.
0	IBSFFV	IBS feature flags valid.

##### **CPUID Fn8000\_001B\_E[D,C,B]X Reserved**

The values returned in EBX, ECX, and EDX are undefined and are reserved.

#### E.4.14 Function 8000\_001Ch—Lightweight Profiling Capabilities

If lightweight profiling (LWP) is supported ([CPUID Fn8000\\_0001\\_ECX](#)[LWP] = 1), this CPUID function can be used to obtain information about LWP features supported by the processor. If LWP is not supported ([CPUID Fn8000\\_0001\\_ECX](#)[LWP] = 0), this function number is reserved. For more information on using LWP, see “Lightweight Profiling” in APM2.

### **CPUID Fn8000\_001C\_EAX Lightweight Profiling Capabilities 0**

The value returned in EAX provides the following information about LWP capabilities supported by the processor:

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31	Lwplnt	Interrupt on threshold overflow available.
30	LwpPTSC	Performance time stamp counter in event record is available.
29	LwpCont	Sampling in continuous mode is available.
28:7	—	Reserved.
6	LwpRNH	Core reference clocks not halted event available.
5	LwpCNH	Core clocks not halted event available.
4	LwpDME	DC miss event available.
3	LwpBRE	Branch retired event available.
2	LwpIRE	Instructions retired event available.
1	LwpVAL	LWPVAL instruction available.
0	LwpAvail	The LWP feature is available.

### **CPUID Fn8000\_001C\_EBX Lightweight Profiling Capabilities 0**

The value returned in EBX provides the following additional information about LWP capabilities supported by the processor:

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:24	LwpEventOffset	Offset in bytes from the start of the LWPCB to the EventInterval1 field.
23:16	LwpMaxEvents	Maximum EventId value supported.
15:8	LwpEventSize	Event record size. Size in bytes of an event record in the LWP event ring buffer.
7:0	LwpCbSize	Control block size. Size in quadwords of the LWPCB.

### **CPUID Fn8000\_001C\_ECX Lightweight Profiling Capabilities 0**

The value returned in ECX provides the following additional information about LWP capabilities supported by the processor:

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31	LwpCacheLatency	Cache latency filtering supported. Cache-related events can be filtered by latency.
30	LwpCacheLevels	Cache level filtering supported. Cache-related events can be filtered by the cache level that returned the data.
29	LwpIpFiltering	IP filtering supported.
28	LwpBranchPrediction	Branch prediction filtering supported. Branches Retired events can be filtered based on whether the branch was predicted properly.

Bits	Field Name	Description
27:24	—	Reserved.
23:16	LwpMinBufferSize	Event ring buffer size. Minimum size of the LWP event ring buffer, in units of 32 event records.
15:9	LwpVersion	Version of LWP implementation.
8:6	LwpLatencyRnd	Amount by which cache latency is rounded.
5	LwpDataAddress	Data cache miss address valid. Address is valid for cache miss event records.
4:0	LwpLatencyMax	Latency counter size. Size in bits of the cache latency counters.

### **CPUID Fn8000\_001C\_EDX Lightweight Profiling Capabilities 0**

The value returned in EDX provides the following additional information about LWP capabilities supported by the processor:

Bits	Field Name	Description
31	Lwplnt	Interrupt on threshold overflow supported.
30	LwpPTSC	Performance time stamp counter in event record is supported.
29	LwpCont	Sampling in continuous mode is supported.
28:7	—	Reserved.
6	LwpRNH	Core reference clocks not halted event is supported.
5	LwpCNH	Core clocks not halted event is supported.
4	LwpDME	DC miss event is supported.
3	LwpBRE	Branch retired event is supported.
2	LwpIRE	Instructions retired event is supported.
1	LwpVAL	LWPVAL instruction is supported.
0	LwpAvail	Lightweight profiling is supported.

### **E.4.15 Function 8000\_001Dh—Cache Topology Information**

CPUID Fn8000\_001D\_E[D,C,B,A]X reports cache topology information for the cache enumerated by the value passed to the instruction in ECX, referred to as Cache *n* in the following description. To gather information for all cache levels, software must repeatedly execute CPUID with 8000\_001Dh in EAX and ECX set to increasing values beginning with 0 until a value of 00h is returned in the field CacheType (EAX[4:0]) indicating no more cache descriptions are available for this processor.

If CPUID Fn8000\_0001\_ECX[TopologyExtensions] = 0, then CPUID Fn8000\_001Dh is reserved. The value of ECX that returns the null CacheType is represented by *N*. Executing CPUID Fn8000\_001D with ECX = *N* + 1 results in an #UD exception. See .

**CPUID Fn8000\_001D\_EAX\_x[N:0] Cache Properties**

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:26	—	Reserved.
25:14	NumSharingCache	Specifies the number of cores sharing the cache enumerated by $n$ , the value passed to the instruction in ECX. The number of cores sharing this cache is the value of this field incremented by 1.
13:10	—	Reserved.
9	FullyAssociative	Fully associative cache. When set, indicates that the cache is fully associative. If 0 is returned in this field, the cache is set associative.
8	SelfInitialization	Self-initializing cache. When set, indicates that the cache is self initializing; software initialization not required. If 0 is returned in this field, hardware does not initialize this cache.
7:5	CacheLevel	Cache level. Identifies the level of this cache. Note that the enumeration value is not necessarily equal to the cache level. Bits                  Description 000b                Reserved. 001b                Level 1 010b                Level 2 011b                Level 3 111b-100b           Reserved.
4:0	CacheType	Cache type. Identifies the type of cache. Bits                  Description 00h                 Null; no more caches. 01h                 Data cache 02h                 Instruction cache 03h                 Unified cache 1Fh-04h             Reserved.

**CPUID Fn8000\_001D\_EBX\_x[N:0] Cache Properties**See [CPUID Fn8000\\_001D\\_EAX\\_x\[N:0\]](#).

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:22	CacheNumWays	Number of ways for this cache. The number of ways is the value returned in this field incremented by 1.
21:12	CachePhysPartitions	Number of physical line partitions. The number of physical line partitions is the value returned in this field incremented by 1.
11:0	CacheLineSize	Cache line size. The cache line size in bytes is the value returned in this field incremented by 1.

**CPUID Fn8000\_001D\_ECX\_x[N:0] Cache Properties**See [CPUID Fn8000\\_001D\\_EAX\\_x\[N:0\]](#).

Bits	Field Name	Description
31:0	CacheNumSets	Number of ways for set associative cache. Number of ways is the value returned in this field incremented by 1. Only valid for caches that are not fully associative ( <a href="#">Fn8000_001D_EAX_xn[FullyAssociative]</a> = 0).

### **CPUID Fn8000\_001D\_EDX\_x[N:0] Cache Properties**

See [CPUID Fn8000\\_001D\\_EAX\\_x\[N:0\]](#).

Bits	Field Name	Description
31:2	—	Reserved.
1	CachelInclusive	Cache inclusivity. A value of 0 indicates that this cache is not inclusive of lower cache levels. A value of 1 indicates that the cache is inclusive of lower cache levels.
0	WBINVD	Write-Back Invalidate/Invalidate execution scope. A value of 0 returned in this field indicates that the WBINVD/INVND instruction invalidates all lower level caches of non-originating cores sharing this cache. When set, this field indicates that the WBINVD/INVND instruction is not guaranteed to invalidate all lower level caches of non-originating cores sharing this cache.

### **E.4.16 Function 8000\_001Eh—Processor Topology Information**

#### **CPUID Fn8000\_001E\_EAX Extended APIC ID**

If [CPUID Fn8000\\_0001\\_ECX\[TopologyExtensions\]](#) = 0, this function number is reserved.

Bits	Field Name	Description
31:0	ExtendedApicId	Extended APIC ID. If <a href="#">MSR0000_001B[ApicEn]</a> = 0, this field is reserved..

#### **CPUID Fn8000\_001E\_EBX Compute Unit Identifiers**

See [CPUID Fn8000\\_001E\\_EAX](#).

Bits	Field Name	Description
31:16	—	Reserved.
15:8	ThreadsPerComputeUnit	Threads per compute unit (zero-based count). The actual number of cores per compute unit is the value of this field + 1.
7:0	ComputeUnitId	Compute unit ID. Identifies the processor compute unit ID.

**CPUID Fn8000\_001E(ECX) Node Identifiers**

See [CPUID Fn8000\\_001E\\_EAX](#).

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:0	—	Reserved.
10:8	NodesPerProcessor	Specifies the number of nodes in the processor (package/socket) in which this core resides. Node in this context corresponds to a processor die. Encoding is N-1, where N is the number of nodes present in the socket.
7:0	Nodeld	Specifies the ID of the node containing the current core. Nodeld values are unique across the system..

**E.4.17 CPUID Fn8000\_001f—Encrypted Memory Capabilities****CPUID Fn8000\_001F\_EAX**

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
0	SME	Secure Memory Encryption supported
1	SEV	Secure Encrypted Virtualization supported
2	PageFlushMsr	Page Flush MSR available
3	SEV-ES	SEV Encrypted State supported

**CPUID Fn8000\_001F\_EBX**

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
11:6	PhysAddrReduction	Physical Address bit reduction
5:0	CbitPosition	C-bit location in page table entry

**CPUID Fn8000\_001F(ECX)**

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:0	NumEncryptedGuests	Number of encrypted guests supported simultaneously

**CPUID Fn8000\_001F\_EDX**

<b>Bits</b>	<b>Field Name</b>	<b>Description</b>
31:0	MinSevNoEsAsid	Minimum ASID value for an SEV enabled, SEV-ES disabled guest

## E.5 Multiple Core Calculation

Operating systems use one of two possible methods to calculate the number of cores per processor (NC), and the maximum number of cores per processor (MNC). The extended method is recommended, but a legacy method is also available for existing operating systems.

### E.5.1 Legacy Method

The CPUID identification of total number of cores per processor (c) is derived from information returned by the following fields:

- CPUID Fn0000\_0001\_EBX[LogicalProcessorCount]
- CPUID Fn0000\_0001\_EDX[HTT] (Hyper-Threading Technology)
- CPUID Fn8000\_0001\_ECX[CmpLegacy]
- CPUID Fn8000\_0008(ECX[NC] (number of cores - 1)

Table E-5 defines LogicalProcessorCount, HTT, CmpLegacy, and NC as a function of the number of cores per processor (c).

When HTT = 0, LogicalProcessorCount is reserved and the processor contains one core.

When HTT = 1 and CmpLegacy = 1, LogicalProcessorCount represents the number of cores per processor (c).

**Table E-5. LogicalProcessorCount, CmpLegacy, HTT, and NC**

Cores per Processor (c)	CmpLegacy	HTT	LogicalProcessorCount	NC
1	0	0	Reserved	0
2 or more	1	1	c	c-1

The use of CmpLegacy and LogicalProcessorCount for the determination of the number of cores is deprecated. Instead, use NC to determine the number of cores.

### E.5.2 Extended Method (Recommended)

The CPUID identification of total number of cores per processor is derived from information returned by the CPUID Fn8000\_0008(ECX[ApicIdCoreIdSize[3:0]]). This field indicates the number of least significant bits in the CPUID Fn0000\_0001\_EBX[LocalApicId] that indicates core ID within the processor. The size of this field determines the maximum number of cores (MNC) that the processor could theoretically support, not the actual number of cores that are actually implemented or enabled on the processor, as indicated by CPUID Fn8000\_0008(ECX[NC]).

A value of zero for ApicIdCoreIdSize[3:0] indicates that the legacy method (section 2.1) should be used to derive the maximum number of cores:

$$\text{MNC} = \text{CPUID Fn8000_0008(ECX[NC])} + 1.$$

For non-zero values of ApicIdCoreIdSize[3:0],

$$\text{MNC} = (2^{\text{ApicIdCoreIdSize}[3:0]})$$

**APIC Enumeration Requirements.** System hardware and system firmware must ensure that the maximum number of cores per processor (MNC) exposed to the operating system across all cores and processors in the system is identical.

Local ApicId MNC rule: The ApicId of core j on processor node i must be enumerated/assigned as:

$$\text{LocalApicId}[\text{proc}=i, \text{core}=j] = (\text{OFFSET\_IDX} + i) * \text{MNC} + j$$

Where "OFFSET\_IDX" is an integer offset (0 to N) used to shift up the core LocalApicId values to allow room for IOAPIC devices. This assignment allows software to use a simple bitmask in addressing all the cores of a single processor. (The assignment also has the effect of reserving some IDs from use to ensure alignment of the ID of core 0 on each processor.)

For example, consider a 3-processor system where:

processor 0 has 4 cores

processor 1 has 1 core

processor 2 has 2 cores

there are 8 IOAPIC devices

cpuid.core\_id\_bits = 2 for all cases, so MNC=4

The LocalApicId and IOAPIC ID spaces cannot be disjointed and must be enumerated in the same ID space in order to support legacy operating systems. Each core can support an 8-bit ApicId. But if each IOAPIC device supports only a 4-bit IOAPIC ID, then the problem can be solved by shifting the LocalApicId space to start at some integer multiple of MNC, such as offset 8 (MNC = 4; OFFSET\_IDX=2):

$$\text{LocalApicId}[\text{proc}=0, \text{core}=0] = (2+0)*4 + 0 = 0x08$$

$$\text{LocalApicId}[\text{proc}=0, \text{core}=1] = (2+0)*4 + 1 = 0x09$$

$$\text{LocalApicId}[\text{proc}=0, \text{core}=2] = (2+0)*4 + 2 = 0x0A$$

$$\text{LocalApicId}[\text{proc}=0, \text{core}=3] = (2+0)*4 + 3 = 0x0B$$

$$\text{LocalApicId}[\text{proc}=1, \text{core}=0] = (2+1)*4 + 0 = 0x0C$$

LocalApicId 0xD to 0xF are reserved

$$\text{LocalApicId}[\text{proc}=2, \text{core}=0] = (2+2)*4 + 0 = 0x10$$

$$\text{LocalApicId}[\text{proc}=2, \text{core}=1] = (2+2)*4 + 1 = 0x11$$

LocalApicId 0x12 and 0x13 are reserved

It is recommended that system firmware use the following LocalApicId assignments for the broadest operating system support. Given N = (Number\_Of\_Processors \* MNC) and M = Number\_Of\_IOAPICs:

- If (N+M) < 16, assign the LocalApicIds for the cores first from 0 to N-1, and the IOAPIC IDs from N to

$N+(M-1)$ .

- If  $(N+M) \geq 16$ , assign the IOAPIC IDs first from 0 to  $M-1$ , and the LocalApicIds for the cores from  $K$  to  $K+(N-1)$ , where  $K$  is an integer multiple of MNC greater than  $M-1$ .



## Appendix F Instruction Effects on RFLAGS

The flags in the RFLAGS register are described in “Flags Register” in Volume 1 and “RFLAGS Register” in Volume 2. Table F-1 summarizes the effect that instructions have on these flags. The table includes all instructions that affect the flags. Instructions not shown have no effect on RFLAGS.

The following codes are used within the table:

- 0—The flag is always cleared to 0.
- 1—The flag is always set to 1.
- AH—The flag is loaded with value from AH register.
- Mod—The flag is modified, depending on the results of the instruction.
- Pop—The flag is loaded with value popped off of the stack.
- Tst—The flag is tested.
- U—The effect on the flag is undefined.
- Gray shaded cells indicate that the flag is not affected by the instruction.

**Table F-1. Instruction Effects on RFLAGS**

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																	
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0	
AAA									U					U	U	Tst Mod	U	Mod
AAS									U					Mod	Mod	U	Mod	U
AAD									Mod				Mod	Mod	Mod	Mod	Tst Mod	
AAM									Mod				Mod	Mod	Mod	Mod		
ADC									Mod				Mod	Mod	Mod	Mod	Tst Mod	
ADD									Mod				Mod	Mod	Mod	Mod		Mod
AND									0				Mod	Mod	U	Mod	0	
ARPL														Mod				
BSF									U				U	Mod	U	U	U	
BSR																		
BT									U					U	U	U	U	Mod
BTC																		
BTR																		
BTS																		
BZHI									0				Mod	Mod	U	U	Mod	
CLC																		0
CLD										0								
CLI		Mod						TST			Mod							
CMC																		Mod
CMOVcc									Tst					Tst	Tst		Tst	Tst
CMP									Mod				Mod	Mod	Mod	Mod	Mod	

**Table F-1. Instruction Effects on RFLAGS (continued)**

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
CMPSx									Mod	Tst			Mod	Mod	Mod	Mod	Mod
CMPXCHG									Mod				Mod	Mod	Mod	Mod	Mod
CMPXCHG8B													Mod				
CMPXCHG16B													Mod				
COMISD COMISS									0				0	Mod	0	Mod	Mod
DAA DAS									U				Mod	Mod	Tst Mod	Mod	Tst Mod
DEC									Mod				Mod	Mod	Mod	Mod	
DIV									U				U	U	U	U	U
FCMOVcc													Tst		Tst	Tst	
FCOMI FCOMIP FUCOMI FUCOMIP													Mod		Mod	Mod	
IDIV									U				U	U	U	U	U
IMUL									Mod				U	U	U	U	Mod
INC									Mod				Mod	Mod	Mod	Mod	
IN									Tst								
INSx									Tst		Tst						
INT INT 3			Mod	Mod	Tst Mod	0	Mod	Tst			Mod	0					
INTO				Mod	Tst Mod	0	Mod	Tst	Tst		Mod	Mod					
IRET <sub>x</sub>	Pop	Pop	Pop	Pop	Tst Pop	Pop	Tst Pop	Tst Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop
Jcc									Tst				Tst	Tst		Tst	Tst
LAR													Mod				
LODSx										Tst							
LOOPE LOOPNE													Tst				
LSL													Mod				
LZCNT									U				U	Mod	U	U	Mod
MOVSx										Tst							
MUL									Mod				U	U	U	U	Mod
NEG									Mod				Mod	Mod	Mod	Mod	Mod
OR									0				Mod	Mod	U	Mod	0
OUT									Tst								
OUTSx									Tst		Tst						
POPCNT									0				0	Mod	0	0	0
POP <sub>F</sub> x	Pop	Tst	Mod	Pop	Tst	0	Pop	Tst Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop

**Table F-1. Instruction Effects on RFLAGS (continued)**

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
RCL 1									Mod								Tst Mod
RCL count									U								Tst Mod
RCR 1									Mod								Tst Mod
RCR count									U								Tst Mod
ROL 1									Mod								Mod
ROL count									U								Mod
ROR 1									Mod								Mod
ROR count									U								Mod
RSM	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod
SAHF											AH	AH	AH	AH	AH		
SHL/SAL 1								Mod				Mod	Mod	U	Mod	Mod	
SHL/SAL count								U				Mod	Mod	U	Mod	Mod	
SAR 1								Mod				Mod	Mod	U	Mod	Mod	
SAR count								U				Mod	Mod	U	Mod	Mod	
SBB								Mod				Mod	Mod	Mod	Mod		Tst Mod
SCASx								Mod	Tst			Mod	Mod	Mod	Mod	Mod	
SETcc								Tst				Tst	Tst		Tst	Tst	
SHLD 1 SHRD 1								Mod				Mod	Mod	U	Mod	Mod	
SHLD count SHRD count								U				Mod	Mod	U	Mod	Mod	
SHR 1								Mod				Mod	Mod	U	Mod	Mod	
SHR count								U				Mod	Mod	U	Mod	Mod	
STC																	1
STD										1							
STI		Mod						Tst			Mod						
STOSx										Tst							
SUB								Mod				Mod	Mod	Mod	Mod	Mod	
SYSCALL	Mod	Mod	Mod	Mod	0	0	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod
SYSENTER					0	0					0						
SYSRET	Mod	Mod	Mod	Mod		0	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod
TEST									0				Mod	Mod	U	Mod	0
UCOMISD UCOMISS									0				0	Mod	0	Mod	Mod

**Table F-1. Instruction Effects on RFLAGS (continued)**

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
VERR															Mod		
VERW																	
XADD									Mod					Mod	Mod	Mod	Mod
XOR									0					Mod	Mod	U	Mod

# Index

---

## Symbols

#VMEXIT..... 439

## Numerics

0F\_38h opcode map..... 462

0F\_3Ah opcode map..... 462

16-bit mode..... xxiii

32-bit mode..... xxiii

64-bit mode..... xxiii

## A

AAA..... 73

AAD..... 74

AAM..... 75

AAS..... 76

ADC..... 77

ADD..... 79, 80

address size prefix ..... 9, 25

addressing

  byte registers..... 26

  effective address..... 491, 494, 495, 497

  PC-relative..... 24

  RIP-relative ..... xxviii, 24

AMD64 Instruction-set Architecture ..... 531

AMD64 ISA..... 531

AND..... 83

ANDN..... 85

ARPL ..... 355

## B

base field..... 496, 497

BEXTR (immediate form)..... 89

BEXTR (register form) ..... 87

biased exponent ..... xxiv

BLCFILL..... 91

BLCI ..... 93

BLCIC..... 95

BLCMSK..... 97

BLCS ..... 99

BLSFILL ..... 101

BLSI..... 103

BLSIC ..... 105

BLSMSK ..... 107

BLSR ..... 109

BOUND..... 111

BSF ..... 113

BSR..... 114

BSWAP ..... 115

BT ..... 116

BTC..... 118

BTR..... 120

BTS ..... 122

byte register addressing ..... 26

BZHI ..... 124

## C

CALL ..... 15

  far call ..... 128

  near call ..... 126

CBW..... 135

CDQ..... 136

CDQE..... 135

CLC..... 137

CLD..... 138

CLFLUSH..... 139

CLGI ..... 358

CLI..... 359

CLTS ..... 361

CMC..... 144

CMOVcc..... 145, 458

CMP..... 149

CMPSX..... 152

CMPXCHG..... 154

CMPXCHG16B..... 156

CMPXCHG8B..... 156

commit ..... xxiv

compatibility mode ..... xxiv

condition codes

  rFLAGS ..... 458, 479

count ..... 499

CPUID ..... 158

  extended functions ..... 158

  feature flags ..... 534

  standard functions ..... 158

CPUID instruction

  testing for ..... 158

CQO ..... 136

CRC32 ..... 160

CWD ..... 136

CWDE ..... 135

## D

DAA ..... 162

DAS ..... 163

data types

  128-bit media ..... 44

64-bit media.....	48
general-purpose.....	40
x87.....	50
DEC .....	16, 164, 527
direct referencing.....	xxiv
displacements.....	xxiv, 24
DIV .....	166
double quadword .....	xxiv
doubleword .....	xxiv
<b>E</b>	
eAX–eSP register .....	xxx
effective address .....	491, 494, 495, 497
effective address size.....	xxv
effective operand size.....	xxv
eFLAGS register.....	xxx
eIP register.....	xxx
element .....	xxv
endián order .....	xxxii, 4
ENTER.....	15, 168
exceptions .....	xxv, 51
exponent .....	xxiv
<b>F</b>	
FCMOVcc .....	479
flush .....	xxv
<b>G</b>	
general-purpose registers.....	38
<b>H</b>	
HLT .....	362
<b>I</b>	
IDIV .....	170
IGN .....	xxv
immediate operands .....	24, 499
IMUL .....	172
IN.....	174
INC .....	16, 176, 527
index field.....	497
indirect .....	xxv
INSB .....	178
INSD .....	178
instruction opcode.....	16
instructions	
128-bit & 256-bit media.....	537
64-bit media.....	537
effects on rFLAGS .....	637
encoding syntax.....	1
general-purpose.....	71, 537
invalid in 64-bit mode.....	525
invalid in long mode .....	526
reassigned in 64-bit mode.....	526
SSE .....	537
system .....	353, 537
x87 .....	537
INSW.....	178
INSx.....	178
INT.....	180
INT 3 .....	363
interrupt vectors .....	51
INTO .....	187
INVD.....	366
INVLPG .....	367
INVLPGA .....	368
IRET .....	369
IRETD .....	369
IRETQ .....	369
<b>J</b>	
Jcc .....	15, 188, 458
JCXZ .....	192
JECXZ .....	192
JMP .....	15
far jump .....	195
near jump .....	193
JRCXZ .....	192
JrCXZ .....	15
<b>L</b>	
LAHF .....	200
LAR .....	375
LDS .....	201
LEA .....	203
LEAVE .....	15, 205
legacy mode .....	xxvi
legacy x86 .....	xxvi
LES .....	201
LFENCE .....	206
LFS .....	201
LGDT .....	15, 377
LGS .....	201
LIDT .....	15, 379
LLDT .....	15, 381
LLWPCB .....	207
LMSW .....	383
LOCK prefix .....	11
LODSB .....	210
LODSD .....	210
LODSQ .....	210
LODSW .....	210
LODSx .....	210
long mode .....	xxvi

LOOP .....	15
LOOPcc .....	15
LOOPx .....	212
LSB .....	xxvi
lsb .....	xxvi
LSL .....	384
LSS .....	201
LTR .....	15, 386
LWPINS .....	214
LWPVAL .....	216
LZCNT .....	219
<b>M</b>	
mask .....	xxvi
MBZ .....	xxvi
MFENCE .....	221
mod field .....	494
mode-register-memory (ModRM) .....	490
modes .....	529
16-bit .....	xxiii
32-bit .....	xxiii
64-bit .....	xxiii, 529
compatibility .....	xxiv, 529
legacy .....	xxvi
long .....	xxvi, 529
protected .....	xxvii
real .....	xxviii
virtual-8086 .....	xxix
ModRM .....	490
ModRM byte .....	17, 27, 459, 470, 490
moffset .....	xxvii
MONITOR .....	388
MOV .....	224
MOV CRn .....	15, 390
MOV DRn .....	15, 392
MOVBE .....	227
MOVD .....	229
MOVMSKPD .....	233
MOVMSKPS .....	235
MOVNTI .....	237
MOVS .....	239
MOVSX .....	241
MOVSx .....	239
MOVSXD .....	242
MOVZX .....	243
MSB .....	xxvii
msb .....	xxvii
MSR .....	xxxii
MUL .....	244
multimedia instructions .....	xxvii
MULX .....	246
MWAIT .....	394

**N**

NEG .....	250
NOP .....	252, 528
NOT .....	253
notation .....	53

**O**

octword .....	xxvii
offset .....	xxvii, 24
one-byte opcodes .....	450
opcode .....	16
two-byte .....	452
opcode map .....	
0F_38h .....	462
0F_3Ah .....	462
primary .....	450
secondary .....	452
opcode maps .....	450
opcodes .....	
3DNow!™ .....	467
group 1 .....	459
group 10 .....	461
group 11 .....	461
group 12 .....	461
group 13 .....	461
group 14 .....	461
group 16 .....	462
group 17 .....	462
group 1a .....	460
group 2 .....	460
group 3 .....	460
group 4 .....	460
group 5 .....	460
group 6 .....	461
group 7 .....	461
group 8 .....	461
group 9 .....	461
group P .....	462
groups .....	459
ModRM byte .....	459
one-byte .....	450
x87 opcode map .....	470
operands .....	
immediate .....	24, 499
size .....	7, 499, 500, 526
OR .....	254
OUT .....	257
OUTS .....	258
OUTSB .....	258
OUTSD .....	258
OUTSW .....	258
overflow .....	xxvii

**P**

packed .....	xxvii
PAUSE .....	260
PC-relative addressing.....	24
PDEP.....	261
PEXT .....	263
POP .....	265
POP FS.....	15
POP GS .....	15
POP reg .....	15
POP reg/mem .....	15
POPAD .....	267
POPAX.....	267
POPCNT.....	268
POPF .....	270
POPFD .....	270
POPFQ .....	15, 270
PREFETCH .....	273
PREFETCHlevel .....	275
PREFETCHW .....	273
prefix	
REX .....	14
prefixes	
address size .....	9, 25
LOCK .....	11
operand size .....	7
repeat .....	12
REX .....	25
segment .....	10
primary opcode map .....	450
processor feature identification (rFLAGS.ID) .....	158
processor vendor.....	159
protected mode .....	xxvii
PUSH .....	277
PUSH FS .....	15
PUSH GS.....	15
PUSH imm32 .....	15
PUSH imm8.....	15
PUSH reg.....	15
PUSH reg/mem .....	15
PUSHA.....	279
PUSHAD .....	279
PUSHF .....	280
PUSHFD.....	280
PUSHFQ.....	15, 280

**Q**

quadword .....	xxvii
----------------	-------

**R**

r/m field .....	459
-----------------	-----

r8-r15 .....	xxxii
rAX-rSP .....	xxxii
RAZ.....	xxvii
RCL.....	282
RCR.....	284
RDFSBASE .....	286, 343
RDGSBASE.....	286, 343
RDMSR .....	396
RDPMC .....	397
RDRAND .....	287
RDTSC .....	399
RDTSCP .....	401
real address mode. See real mode .....	
real mode .....	xxviii
reg field.....	459, 491, 493, 494
registers	
eAX-eSP .....	xxx
eFLAGS .....	xxx
eIP.....	xxx
encodings.....	26
general-purpose .....	38
MMX .....	48
r8-r15.....	xxxii
rAX-rSP .....	xxxii
rFLAGS.....	xxxii, 458, 479, 637
rIP .....	xxxii
segment .....	40
system .....	41
x87.....	50
XMM .....	43
relative .....	xxviii
REPx prefixes.....	12
reserved .....	xxviii
RET	
far return .....	290
near return.....	289
RET (Near).....	15
revision history .....	xvii
REX prefix .....	14
REX prefix .....	490
REX prefixes .....	25
REX.B bit .....	24, 56, 494, 496
REX.R bit .....	23, 493
REX.W bit .....	23
REX.X bit .....	24
rFLAGS conditions codes.....	458, 479
rFLAGS register .....	xxxii, 637
rIP register.....	xxxii
RIP-relative addressing.....	xxviii, 24
ROL.....	294
ROR .....	296
RORX.....	298
rotate count.....	499

RSM .....	403	STR .....	416
RSM instruction .....	403	SUB .....	332
<b>S</b>		SWAPGS .....	417
SAHF .....	300	syntax .....	52
SAL .....	301	SYSCALL .....	419
SAR .....	304	SYSENTER .....	423
SARX .....	306	SYSEXIT .....	425
SBB .....	308	SYSRET .....	427
SBZ .....	xxviii	system data structures .....	42
scale field .....	497		
scale-index-base (SIB) .....	490	<b>T</b>	
SCAS .....	310	T1MSKC .....	334
SCASB .....	310	TEST .....	336
SCASD .....	310	three-byte prefix .....	29
SCASQ .....	310	TSS .....	xxix
SCASW .....	310	two-byte opcode .....	452
secondary opcode map .....	452	two-byte prefix .....	32
segment prefixes .....	10, 528	TZCNT .....	338
segment registers .....	40	TZMSK .....	340
set .....	xxviii		
SETcc .....	312, 458	<b>U</b>	
SFENCE .....	314	UD2 .....	342
SGDT .....	405	underflow .....	xxix
shift count .....	499		
SHL .....	301, 315	<b>V</b>	
SHLD .....	316	vector .....	xxix
SHLX .....	318	VERR .....	431
SHR .....	320	VERW .....	433
SHRD .....	322	VEX prefix .....	490
SHRX .....	324	virtual-8086 mode .....	xxix
SIB .....	490	VMLOAD .....	434
SIB byte .....	19, 27, 495	VMMCALL .....	436
SIDT .....	406	VMRUN .....	437
SKINIT .....	407	VMSAVE .....	442
SLDT .....	409		
SLWPCB .....	326	<b>W</b>	
SMSW .....	411	WBINVD .....	444
SSE .....	xxviii	WRMSR .....	222, 248, 445
SSE2 .....	xxix		
SSE3 .....	xxix	<b>X</b>	
STC .....	328	XADD .....	344
STD .....	329	XCHG .....	346
STGI .....	415	XLAT .....	348
STI .....	413	XLATB .....	348
sticky bits .....	xxix	XOP prefix .....	490
STOS .....	330	XOR .....	349
STOSB .....	330		
STOSD .....	330	<b>Z</b>	
STOSQ .....	330	zero-extension .....	499
STOSW .....	330		

