This chapter lists all the instructions in the Intel Architecture instruction set, divided into three functional groups: integer, floating-point, and system. It also briefly describes each of the integer instructions.

Brief descriptions of the floating-point instructions are given in “Floating-Point Unit”; brief descriptions of the system instructions are given in the Intel Architecture Software Developer’s Manual, Volume 3.

Detailed descriptions of all the Intel Architecture instructions are given in Intel Architecture Software Developer’s Manual, Volume 2. Included in this volume are a description of each instruction’s encoding and operation, the effect of an instruction on the EFLAGS flags, and the exceptions an instruction may generate.

30.1 New Intel Architecture Instructions

The following sections give the Intel Architecture instructions that were new in the MMX Technology and in the Pentium Pro, Pentium, and Intel486 processors.

30.1.1 New Instructions Introduced with the MMX™ Technology

The Intel MMX technology introduced a new set of instructions to the Intel Architecture, designed to enhance the performance of multimedia applications. These instructions are recognized by all Intel Architecture processors that implement the MMX technology. The MMX instructions are listed in “MMX™ Technology Instructions”.

30.1.2 New Instructions in the Pentium® Pro Processor

The following instructions are new in the Pentium Pro processor:

- **CMOVcc**—Conditional move (see “Conditional Move Instructions”).
- **FCMOVcc**—Floating-point conditional move on condition-code flags in EFLAGS register (see “Data Transfer Instructions”).
- **FCOM|FCOMI|FUCOM|FUCOMI**—Floating-point compare and set condition-code flags in EFLAGS register (see “Comparison and Classification Instructions”).
- **RDPMC**—Read performance monitoring counters (see “RDPMC—Read Performance-Monitoring Counters” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2). (This instruction is also available in all Pentium® processors that implement the MMX™ technology.)
- **UD2**—Undefined instruction (see “No-Operation and Undefined Instructions”).
30.1.3 New Instructions in the Pentium® Processor

The following instructions are new in the Pentium processor:

- CMPXCHG8B (compare and exchange 8 bytes) instruction.
- CPUID (CPU identification) instruction. (This instruction was introduced in the Pentium® processor and added to later versions of the Intel486™ processor.)
- RDTSC (read time-stamp counter) instruction.
- RDMSR (read model-specific register) instruction.
- WRMSR (write model-specific register) instruction.
- RSM (resume from SMM) instruction.

The form of the MOV instruction used to access the test registers has been removed on the Pentium and future Intel Architecture processors.

30.1.4 New Instructions in the Intel486™ Processor

The following instructions are new in the Intel486 processor:

- BSWAP (byte swap) instruction.
- XADD (exchange and add) instruction.
- CMPXCHG (compare and exchange) instruction.
- INVD (invalidate cache) instruction.
- WBINVD (write-back and invalidate cache) instruction.
- INVLP (invalidate TLB entry) instruction.

30.2 Instruction Set List

This section lists all the Intel Architecture instructions divided into three major groups: integer, MMX technology, floating-point, and system instructions. For each instruction, the mnemonic and descriptive names are given. When two or more mnemonics are given (for example, CMOV/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move is not below or equal) represent the same condition.

30.2.1 Integer Instructions

Integer instructions perform the integer arithmetic, logic, and program flow control operations that programmers commonly use to write application and system software to run on an Intel Architecture processor. In the following sections, the integer instructions are divided into several instruction subgroups.
### Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>CMOVE/CMOVZ</td>
<td>Conditional move if equal/Conditional move if zero</td>
</tr>
<tr>
<td>CMOVNE/CMOVNZ</td>
<td>Conditional move if not equal/Conditional move if not zero</td>
</tr>
<tr>
<td>CMOVA/CMOVNBE</td>
<td>Conditional move if above/Conditional move if not below or equal</td>
</tr>
<tr>
<td>CMOVAE/CMOVNB</td>
<td>Conditional move if above or equal/Conditional move if not below</td>
</tr>
<tr>
<td>CMOVB/CMOVNAE</td>
<td>Conditional move if below/Conditional move if not above or equal</td>
</tr>
<tr>
<td>CMOVBE/CMOVNA</td>
<td>Conditional move if below or equal/Conditional move if not above</td>
</tr>
<tr>
<td>CMOVG/CMOVNLE</td>
<td>Conditional move if greater/Conditional move if not less or equal</td>
</tr>
<tr>
<td>CMOVGE/CMOVNL</td>
<td>Conditional move if greater or equal/Conditional move if not less</td>
</tr>
<tr>
<td>CMOL/CMOVNGE</td>
<td>Conditional move if less/Conditional move if not greater or equal</td>
</tr>
<tr>
<td>CMOL/CMOVNG</td>
<td>Conditional move if less or equal/Conditional move if not greater</td>
</tr>
<tr>
<td>CMOVC</td>
<td>Conditional move if carry</td>
</tr>
<tr>
<td>CMOVNC</td>
<td>Conditional move if not carry</td>
</tr>
<tr>
<td>CMOVO</td>
<td>Conditional move if overflow</td>
</tr>
<tr>
<td>CMOVNO</td>
<td>Conditional move if not overflow</td>
</tr>
<tr>
<td>CMOVNS</td>
<td>Conditional move if sign (negative)</td>
</tr>
<tr>
<td>CMOVNS</td>
<td>Conditional move if sign (non-negative)</td>
</tr>
<tr>
<td>CMOVVP/CMOVPE</td>
<td>Conditional move if parity/Conditional move if parity even</td>
</tr>
<tr>
<td>CMOVNVP/CMOVPO</td>
<td>Conditional move if not parity/Conditional move if parity odd</td>
</tr>
<tr>
<td>XCHG</td>
<td>Exchange</td>
</tr>
<tr>
<td>BSWAP</td>
<td>Byte swap</td>
</tr>
<tr>
<td>XADD</td>
<td>Exchange and add</td>
</tr>
<tr>
<td>CMPXCHG</td>
<td>Compare and exchange</td>
</tr>
<tr>
<td>CMPXCHGBB</td>
<td>Compare and exchange 8 bytes</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push onto stack</td>
</tr>
<tr>
<td>POP</td>
<td>Pop off of stack</td>
</tr>
<tr>
<td>PUSHA/PUSHAD</td>
<td>Push general-purpose registers onto stack</td>
</tr>
<tr>
<td>POPA/POPAD</td>
<td>Pop general-purpose registers from stack</td>
</tr>
<tr>
<td>IN</td>
<td>Read from a port</td>
</tr>
<tr>
<td>OUT</td>
<td>Write to a port</td>
</tr>
<tr>
<td>CWD/CDQ</td>
<td>Convert word to doubleword/Convert doubleword to quadword</td>
</tr>
<tr>
<td>CBW/CWDE</td>
<td>Convert byte to word/Convert word to doubleword in EAX register</td>
</tr>
<tr>
<td>MOVZX</td>
<td>Move and sign extend</td>
</tr>
<tr>
<td>MOVZX</td>
<td>Move and zero extend</td>
</tr>
</tbody>
</table>
30.2.1.2 Binary Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Integer add</td>
</tr>
<tr>
<td>ADC</td>
<td>Add with carry</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>SBB</td>
<td>Subtract with borrow</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed multiply</td>
</tr>
<tr>
<td>MUL</td>
<td>Unsigned multiply</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed divide</td>
</tr>
<tr>
<td>DIV</td>
<td>Unsigned divide</td>
</tr>
<tr>
<td>INC</td>
<td>Increment</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
</tr>
</tbody>
</table>

30.2.1.3 Decimal Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAA</td>
<td>Decimal adjust after addition</td>
</tr>
<tr>
<td>DAS</td>
<td>Decimal adjust after subtraction</td>
</tr>
<tr>
<td>AAA</td>
<td>ASCII adjust after addition</td>
</tr>
<tr>
<td>AAS</td>
<td>ASCII adjust after subtraction</td>
</tr>
<tr>
<td>AAM</td>
<td>ASCII adjust after multiplication</td>
</tr>
<tr>
<td>AAD</td>
<td>ASCII adjust before division</td>
</tr>
</tbody>
</table>

30.2.1.4 Logic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>And</td>
</tr>
<tr>
<td>OR</td>
<td>Or</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive or</td>
</tr>
<tr>
<td>NOT</td>
<td>Not</td>
</tr>
</tbody>
</table>
30.2.1.5 Shift and Rotate Instructions

- **SAR**  Shift arithmetic right
- **SHR**  Shift logical right
- **SAL/SHL**  Shift arithmetic left/Shift logical left
- **SHRD**  Shift right double
- **SHLD**  Shift left double
- **ROR**  Rotate right
- **ROL**  Rotate left
- **RCR**  Rotate through carry right
- **RCL**  Rotate through carry left

30.2.1.6 Bit and Byte Instructions

- **BT**  Bit test
- **BTS**  Bit test and set
- **BTR**  Bit test and reset
- **BTC**  Bit test and complement
- **BSF**  Bit scan forward
- **BSR**  Bit scan reverse
- **SETE/SETZ**  Set byte if equal/Set byte if zero
- **SETNE/SETNZ**  Set byte if not equal/Set byte if not zero
- **SETA/SETNBE**  Set byte if above/Set byte if not below or equal
- **SETAE/SETNB/SETNC**  Set byte if above or equal/Set byte if not below/Set byte if not carry
- **SETB/SETNAE/SETC**  Set byte if below/Set byte if not above or equal/Set byte if carry
- **SETBE/SETNA**  Set byte if below or equal/Set byte if not above
- **SETG/SETNLE**  Set byte if greater/Set byte if not less or equal
- **SETGE/SETNL**  Set byte if greater or equal/Set byte if not less
- **SETL/SETNGE**  Set byte if less/Set byte if not greater or equal
- **SETLE/SETNG**  Set byte if less or equal/Set byte if not greater
- **SETS**  Set byte if sign (negative)
- **SETNS**  Set byte if not sign (non-negative)
- **SETO**  Set byte if overflow
- **SETNO**  Set byte if not overflow
- **SETPE/SETP**  Set byte if parity even/Set byte if parity
- **SETPO/SETPN**  Set byte if parity odd/Set byte if not parity
- **TEST**  Logical compare
30.2.1.7 Control Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>JE/JZ</td>
<td>Jump if equal/Jump if zero</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>Jump if not equal/Jump if not zero</td>
</tr>
<tr>
<td>JA/JNBE</td>
<td>Jump if above/Jump if not below or equal</td>
</tr>
<tr>
<td>JAE/JNBE</td>
<td>Jump if above or equal/Jump if not below</td>
</tr>
<tr>
<td>JB/JNAE</td>
<td>Jump if below/Jump if not above or equal</td>
</tr>
<tr>
<td>JBE/JNA</td>
<td>Jump if below or equal/Jump if not above</td>
</tr>
<tr>
<td>JG/JNLE</td>
<td>Jump if greater/Jump if not less or equal</td>
</tr>
<tr>
<td>JGE/JNL</td>
<td>Jump if greater or equal/Jump if not less</td>
</tr>
<tr>
<td>JL/JNGE</td>
<td>Jump if less/Jump if not greater or equal</td>
</tr>
<tr>
<td>JLE/JNG</td>
<td>Jump if less or equal/Jump if not greater</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign (negative)</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not sign (non-negative)</td>
</tr>
<tr>
<td>JPO/JNP</td>
<td>Jump if parity odd/Jump if not parity</td>
</tr>
<tr>
<td>JPE/JP</td>
<td>Jump if parity even/Jump if parity</td>
</tr>
<tr>
<td>JCXZ/JECXZ</td>
<td>Jump register CX zero/Jump register ECX zero</td>
</tr>
<tr>
<td>LOOP</td>
<td>Loop with ECX counter</td>
</tr>
<tr>
<td>LOOPZ/LOOPE</td>
<td>Loop with ECX and zero/Loop with ECX and equal</td>
</tr>
<tr>
<td>LOOPNZ/LOOPNE</td>
<td>Loop with ECX and not zero/Loop with ECX and not equal</td>
</tr>
<tr>
<td>CALL</td>
<td>Call procedure</td>
</tr>
<tr>
<td>RET</td>
<td>Return</td>
</tr>
<tr>
<td>IRET</td>
<td>Return from interrupt</td>
</tr>
<tr>
<td>INT</td>
<td>Software interrupt</td>
</tr>
<tr>
<td>INTO</td>
<td>Interrupt on overflow</td>
</tr>
<tr>
<td>BOUND</td>
<td>Detect value out of range</td>
</tr>
<tr>
<td>ENTER</td>
<td>High-level procedure entry</td>
</tr>
<tr>
<td>LEAVE</td>
<td>High-level procedure exit</td>
</tr>
</tbody>
</table>
30.2.1.8 String Instructions

- **MOVS/MOVSB**: Move string/Move byte string
- **MOVS/MOVSW**: Move string/Move word string
- **MOVS/MOVSD**: Move string/Move doubleword string
- **CMPS/CMPSB**: Compare string/Compare byte string
- **CMPS/CMPSW**: Compare string/Compare word string
- **CMPS/CMPSD**: Compare string/Compare doubleword string
- **SCAS/SCASB**: Scan string/Scan byte string
- **SCAS/SCASW**: Scan string/Scan word string
- **SCAS/SCASD**: Scan string/Scan doubleword string
- **LODS/LODSB**: Load string/Load byte string
- **LODS/LODSW**: Load string/Load word string
- **LODS/LODSD**: Load string/Load doubleword string
- **STOS/STOSB**: Store string/Store byte string
- **STOS/STOSW**: Store string/Store word string
- **STOS/STOSD**: Store string/Store doubleword string
- **REP**: Repeat while ECX not zero
- **REPE/REPZ**: Repeat while equal/Repeat while zero
- **REPNE/REPNZ**: Repeat while not equal/Repeat while not zero
- **INS/INSB**: Input string from port/Input byte string from port
- **INS/INSW**: Input string from port/Input word string from port
- **INS/INSD**: Input string from port/Input doubleword string from port
- **OUTS/OUTSB**: Output string to port/Output byte string to port
- **OUTS/OUTSW**: Output string to port/Output word string to port
- **OUTS/OUTSD**: Output string to port/Output doubleword string to port
30.2.1.9 Flag Control Instructions

STC          Set carry flag
CLC          Clear the carry flag
CMC          Complement the carry flag
CLD          Clear the direction flag
STD          Set direction flag
LAHF         Load flags into AH register
SAHF         Store AH register into flags
PUSHF/PUSHFD Push EFLAGS onto stack
POPF/POPFD   Pop EFLAGS from stack
STI          Set interrupt flag
CLI          Clear the interrupt flag

30.2.1.10 Segment Register Instructions

LDS          Load far pointer using DS
LES          Load far pointer using ES
LFS          Load far pointer using FS
LGS          Load far pointer using GS
LSS          Load far pointer using SS

30.2.1.11 Miscellaneous Instructions

LEA          Load effective address
NOP          No operation
UB2          Undefined instruction
XLAT/XLATB   Table lookup translation
CPUID        Processor Identification

30.2.2 MMX™ Technology Instructions

The MMX instructions execute on those Intel Architecture processors that implement the Intel MMX technology. These instructions operate on packed-byte, packed-word, packed-doubleword, and quadword operands. As with the integer instructions, the following list of MMX instructions is divided into subgroups.
30.2.2.1 MMX™ Data Transfer Instructions

- MOVD: Move doubleword
- MOVQ: Move quadword

30.2.2.2 MMX™ Conversion Instructions

- PACKSSWB: Pack words into bytes with signed saturation
- PACKSSDW: Pack doublewords into words with signed saturation
- PACKUSWB: Pack words into bytes with unsigned saturation
- PUNPCKHBW: Unpack high-order bytes from words
- PUNPCKHWD: Unpack high-order words from doublewords
- PUNPCKHDQ: Unpack high-order doublewords from quadword
- PUNPCKLBW: Unpack low-order bytes from words
- PUNPCKLWD: Unpack low-order words from doublewords
- PUNPCKLDQ: Unpack low-order doublewords from quadword

30.2.2.3 MMX™ Packed Arithmetic Instructions

- PADDB: Add packed bytes
- PADDW: Add packed words
- PADDD: Add packed doublewords
- PADDSB: Add packed bytes with saturation
- PADDSD: Add packed doublewords with saturation
- PADDUSB: Add packed unsigned bytes with saturation
- PADDUSW: Add packed unsigned words with saturation
- PSUBB: Subtract packed bytes
- PSUBW: Subtract packed words
- PSUBD: Subtract packed doublewords
- PSUSSB: Subtract packed bytes with saturation
- PSUBSW: Subtract packed words with saturation
- PSUBUSB: Subtract packed unsigned bytes with saturation
- PSUBUSW: Subtract packed unsigned words with saturation
- PMULHW: Multiply packed words and store high result
- PMULLW: Multiply packed words and store low result
- PMADDWD: Multiply and add packed words
### 30.2.4 MMX™ Comparison Instructions

- **PCMPEQB**: Compare packed bytes for equal
- **PCMPEQW**: Compare packed words for equal
- **PCMPEQD**: Compare packed doublewords for equal
- **PCMPGTB**: Compare packed bytes for greater than
- **PCMPGTW**: Compare packed words for greater than
- **PCMPGTD**: Compare packed doublewords for greater than

### 30.2.5 MMX™ Logic Instructions

- **PAND**: Bitwise logical and
- **PANDN**: Bitwise logical and not
- **POR**: Bitwise logical or
- **PXOR**: Bitwise logical exclusive or

### 30.2.6 MMX™ Shift and Rotate Instructions

- **PSLLW**: Shift packed words left logical
- **PSLLD**: Shift packed doublewords left logical
- **PSLLQ**: Shift packed quadword left logical
- **PSRLW**: Shift packed words right logical
- **PSRLD**: Shift packed doublewords right logical
- **PSRLQ**: Shift packed quadword right logical
- **PSRAW**: Shift packed words right arithmetic
- **PSRAD**: Shift packed doublewords right arithmetic

### 30.2.7 MMX™ State Management

- **EMMS**: Empty MMX state

### 30.2.3 Floating-Point Instructions

The floating-point instructions are those that are executed by the processor’s floating-point unit (FPU). These instructions operate on floating-point (real), extended integer, and binary-coded decimal (BCD) operands. As with the integer instructions, the following list of floating-point instructions is divided into subgroups.
30.2.3.1 Data Transfer

- **FLD**: Load real
- **FST**: Store real
- **FSTP**: Store real and pop
- **FILD**: Load integer
- **FIST**: Store integer
- **FISTP**: Store integer and pop
- **FBLD**: Load BCD
- **FBSTP**: Store BCD and pop
- **FXCH**: Exchange registers
- **FCMOVE**: Floating-point conditional move if equal
- **FCMOVNE**: Floating-point conditional move if not equal
- **FCMOVB**: Floating-point conditional move if below
- **FCMOVBE**: Floating-point conditional move if below or equal
- **FCMOVNB**: Floating-point conditional move if not below
- **FCMOVNBE**: Floating-point conditional move if not below or equal
- **FCMOVU**: Floating-point conditional move if unordered
- **FCMOVNU**: Floating-point conditional move if not unordered

30.2.3.2 Basic Arithmetic

- **FADD**: Add real
- **FADDP**: Add real and pop
- **FIADD**: Add integer
- **FSUB**: Subtract real
- **FSUBP**: Subtract real and pop
- **FISUB**: Subtract integer
- **FSUBR**: Subtract real reverse
- **FSUBRP**: Subtract real reverse and pop
- **FISUBR**: Subtract integer reverse
- **FMUL**: Multiply real
- **FMULP**: Multiply real and pop
- **FIMUL**: Multiply integer
- **FDIV**: Divide real
- **FDIVP**: Divide real and pop
- **FIDIV**: Divide integer
- **FDIVR**: Divide real reverse
- **FDIVRP**: Divide real reverse and pop
- **FIDIVR**: Divide integer reverse
- **FPREM**: Partial remainder
Instruction Set Summary

FPREMI  IEEE Partial remainder
FABS    Absolute value
FCHS    Change sign
FRNDINT Round to integer
FSCALE  Scale by power of two
FSQRT   Square root
FXTRACT Extract exponent and significand

30.2.3.3 Comparison

FCOM    Compare real
FCOMP   Compare real and pop
FCOMPP  Compare real and pop twice
FUCOM   Unordered compare real
FUCOMP  Unordered compare real and pop
FUCOMPP Unordered compare real and pop twice
FICOM   Compare integer
FICOMP  Compare integer and pop
FCOMI   Compare real and set EFLAGS
FUCOMI  Unordered compare real and set EFLAGS
FCOMIP  Compare real, set EFLAGS, and pop
FUCOMIP Unordered compare real, set EFLAGS, and pop
FTST    Test real
FXAM    Examine real

30.2.3.4 Transcendental

FSIN    Sine
FCOS    Cosine
FSIN COS Sine and cosine
FPTAN   Partial tangent
FPATAN  Partial arctangent
F2XM1   $2^x - 1$
FYL2X   $y = \log_2 x$
FYL2XP1 $y = \log_2(x+1)$
30.2.3.5 Load Constants

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD1</td>
<td>Load +1.0</td>
</tr>
<tr>
<td>FLDZ</td>
<td>Load +0.0</td>
</tr>
<tr>
<td>FLDPI</td>
<td>Load $\pi$</td>
</tr>
<tr>
<td>FLDL2E</td>
<td>Load $\log_2 e$</td>
</tr>
<tr>
<td>FLDLN2</td>
<td>Load $\log_2 2$</td>
</tr>
<tr>
<td>FLDL2T</td>
<td>Load $\log_2 10$</td>
</tr>
<tr>
<td>FLDLG2</td>
<td>Load $\log_{10} 2$</td>
</tr>
</tbody>
</table>

30.2.3.6 FPU Control

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINCSTP</td>
<td>Increment FPU register stack pointer</td>
</tr>
<tr>
<td>FDECSTP</td>
<td>Decrement FPU register stack pointer</td>
</tr>
<tr>
<td>FFREE</td>
<td>Free floating-point register</td>
</tr>
<tr>
<td>FINIT</td>
<td>Initialize FPU after checking error conditions</td>
</tr>
<tr>
<td>FFINIT</td>
<td>Initialize FPU without checking error conditions</td>
</tr>
<tr>
<td>FCLEX</td>
<td>Clear floating-point exception flags after checking for error conditions</td>
</tr>
<tr>
<td>FCLEX</td>
<td>Clear floating-point exception flags without checking for error conditions</td>
</tr>
<tr>
<td>FSTCW</td>
<td>Store FPU control word after checking error conditions</td>
</tr>
<tr>
<td>FNSTCW</td>
<td>Store FPU control word without checking error conditions</td>
</tr>
<tr>
<td>FLDCW</td>
<td>Load FPU control word</td>
</tr>
<tr>
<td>FSTENV</td>
<td>Store FPU environment after checking error conditions</td>
</tr>
<tr>
<td>FNSTENV</td>
<td>Store FPU environment without checking error conditions</td>
</tr>
<tr>
<td>FLDENV</td>
<td>Load FPU environment</td>
</tr>
<tr>
<td>FSAVE</td>
<td>Save FPU state after checking error conditions</td>
</tr>
<tr>
<td>FNSAVE</td>
<td>Save FPU state without checking error conditions</td>
</tr>
<tr>
<td>FRSTOR</td>
<td>Restore FPU state</td>
</tr>
<tr>
<td>FSTSW</td>
<td>Store FPU status word after checking error conditions</td>
</tr>
<tr>
<td>FNSTSW</td>
<td>Store FPU status word without checking error conditions</td>
</tr>
<tr>
<td>WAIT/FWAIT</td>
<td>Wait for FPU</td>
</tr>
<tr>
<td>FNOP</td>
<td>FPU no operation</td>
</tr>
</tbody>
</table>
30.2.4 System Instructions

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

- LGDT: Load global descriptor table (GDT) register
- SGDT: Store global descriptor table (GDT) register
- LLDT: Load local descriptor table (LDT) register
- SLDT: Store local descriptor table (LDT) register
- LTR: Load task register
- STR: Store task register
- LIDT: Load interrupt descriptor table (IDT) register
- SIDT: Store interrupt descriptor table (IDT) register
- MOV: Load and store control registers
- LMSW: Load machine status word
- SMSW: Store machine status word
- CLTS: Clear the task-switched flag
- ARPL: Adjust requested privilege level
- LAR: Load access rights
- LSL: Load segment limit
- VERR: Verify segment for reading
- VERW: Verify segment for writing
- MOV: Load and store debug registers
- INVD: Invalidate cache, no writeback
- WBINVD: Invalidate cache, with writeback
- INVLPG: Invalidate TLB Entry
- LOCK (prefix): Lock Bus
- HLT: Halt processor
- RSM: Return from system management mode (SSM)
- RDMSR: Read model-specific register
- WRMSR: Write model-specific register
- RDPMC: Read performance monitoring counters
- RDTSC: Read time stamp counter

30.3 Data Movement Instructions

The data movement instructions move bytes, words, doublewords, or quadwords both between memory and the processor’s registers and between registers. These instructions are divided into four groups:

- General-purpose data movement.
- Exchange.
• Stack manipulation.
• Type-conversion.

30.3.1 General-Purpose Data Movement Instructions

The MOV (move) and CMOVcc (conditional move) instructions transfer data between memory and registers or between registers.

30.3.1.1 Move Instruction

The MOV instruction performs basic load data and store data operations between memory and the processor’s registers and data movement operations between registers. It handles data transfers along the paths listed in Table 30-1. (See “MOV—Move to/from Control Registers” and “MOV—Move to/from Debug Registers” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2, for information on moving data to and from the control and debug registers.)

The MOV instruction cannot move data from one memory location to another or from one segment register to another segment register. Memory-to-memory moves can be performed with the MOVS (string move) instruction (see “String Operations”).

30.3.1.2 Conditional Move Instructions

The CMOVcc instructions are a group of instructions that check the state of the status flags in the EFLAGS register and perform a move operation if the flags are in a specified state (or condition). These instructions can be used to move a 16- or 32-bit value from memory to a general-purpose register or from one general-purpose register to another. The flag state being tested for each instruction is specified with a condition code (cc) that is associated with the instruction. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

Table 30-1. Move Instruction Operations

<table>
<thead>
<tr>
<th>Type of Data Movement</th>
<th>Source → Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>From memory to a register</td>
<td>Memory location → General-purpose register</td>
</tr>
<tr>
<td></td>
<td>Memory location → Segment register</td>
</tr>
<tr>
<td>From a register to memory</td>
<td>General-purpose register → Memory location</td>
</tr>
<tr>
<td></td>
<td>Segment register → Memory location</td>
</tr>
<tr>
<td>Between registers</td>
<td>General-purpose register → General-purpose register</td>
</tr>
<tr>
<td></td>
<td>General-purpose register → Segment register</td>
</tr>
<tr>
<td></td>
<td>Segment register → General-purpose register</td>
</tr>
<tr>
<td></td>
<td>General-purpose register → Control register</td>
</tr>
<tr>
<td></td>
<td>Control register → General-purpose register</td>
</tr>
<tr>
<td></td>
<td>General-purpose register → Debug register</td>
</tr>
<tr>
<td></td>
<td>Debug register → General-purpose register</td>
</tr>
<tr>
<td>Immediate data to a register</td>
<td>Immediate → General-purpose register</td>
</tr>
<tr>
<td>Immediate data to memory</td>
<td>Immediate → Memory location</td>
</tr>
</tbody>
</table>
Table 30-4 shows the mnemonics for the CMOVcc instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letters “CMOV” to form the mnemonics for the CMOVcc instructions. The instructions listed in Table 30-4 as pairs (for example, CMOVA/CMOVNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The CMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF statements and the possibility of branch mispredictions by the processor.

These instructions may not be supported on some processors in the Pentium Pro processor family. Software can check if the CMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).

### 30.3.1.3 Exchange Instructions

The exchange instructions swap the contents of one or more operands and, in some cases, performs additional operations such as asserting the LOCK signal or modifying flags in the EFLAGS register.

The XCHG (exchange) instruction swaps the contents of two operands. This instruction takes the place of three MOV instructions and does not require a temporary location to save the contents of one operand location while the other is being loaded. When a memory operand is used with the XCHG instruction, the processor’s LOCK signal is automatically asserted. This instruction is thus useful for implementing semaphores or similar data structures for process synchronization. (See “Bus Locking” in Chapter 7 of the Intel Architecture Software Developer’s Manual, Volume 3, for more information on bus locking.)

The BSWAP (byte swap) instruction reverses the byte order in a 32-bit register operand. Bit positions 0 through 7 are exchanged with 24 through 31, and bit positions 8 through 15 are exchanged with 16 through 23. Executing this instruction twice in a row leaves the register with the same value as before. The BSWAP instruction is useful for converting between “big-endian” and “little-endian” data formats. This instruction also speeds execution of decimal arithmetic. (The XCHG instruction can be used two swap the bytes in a word.)

### Table 30-2. Conditional Move Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Status Flag States</th>
<th>Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned Conditional Moves</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMOVA/CMOVNBE</td>
<td>(CF or ZF)=0</td>
<td>Above/not below or equal</td>
</tr>
<tr>
<td>CMOVAE/CMOVNBE</td>
<td>CF=0</td>
<td>Above or equal/not below</td>
</tr>
<tr>
<td>CMOVNC</td>
<td>CF=0</td>
<td>Not carry</td>
</tr>
<tr>
<td>CMOVB/CMOVNAE</td>
<td>CF=1</td>
<td>Below/not above or equal</td>
</tr>
<tr>
<td>CMOVC</td>
<td>CF=1</td>
<td>Carry</td>
</tr>
<tr>
<td>CMOVBE/CMOVNA</td>
<td>(CF or ZF)=1</td>
<td>Below or equal/not above</td>
</tr>
<tr>
<td>MOVE/CMOVZ</td>
<td>ZF=1</td>
<td>Equal/zero</td>
</tr>
<tr>
<td>CMOVNE/CMOVNZ</td>
<td>ZF=0</td>
<td>Not equal/not zero</td>
</tr>
<tr>
<td>CMOVPO/CMOVPE</td>
<td>PF=1</td>
<td>Parity/parity even</td>
</tr>
<tr>
<td>CMOVNP/CMOVPO</td>
<td>PF=0</td>
<td>Not parity/parity odd</td>
</tr>
</tbody>
</table>
The XADD (exchange and add) instruction swaps two operands and then stores the sum of the two operands in the destination operand. The status flags in the EFLAGS register indicate the result of the addition. This instruction can be combined with the LOCK prefix (see “LOCK—Assert LOCK# Signal Prefix” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*) in a multiprocessing system to allow multiple processors to execute one DO loop.

The CMPXCHG (compare and exchange) and CMPXCHG8B (compare and exchange 8 bytes) instructions are used to synchronize operations in systems that use multiple processors. The CMPXCHG instruction requires three operands: a source operand in a register, another source operand in the EAX register, and a destination operand. If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register). Otherwise, the original value of the destination operand is loaded in the EAX register. The status flags in the EFLAGS register reflect the result that would have been obtained by subtracting the destination operand from the value in the EAX register.

The CMPXCHG instruction is commonly used for testing and modifying semaphores. It checks to see if a semaphore is free. If the semaphore is free it is marked allocated, otherwise it gets the ID of the current owner. This is all done in one uninterruptible operation. In a single-processor system, the CMPXCHG instruction eliminates the need to switch to protection level 0 (to disable interrupts) before executing multiple instructions to test and modify a semaphore. For multiple processor systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically. (See “Locked Atomic Operations” in Chapter 7 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for more information on atomic operations.)

The CMPXCHG8B instruction also requires three operands: a 64-bit value in EDX:EAX, a 64-bit value in ECX:EBX, and a destination operand in memory. The instruction compares the 64-bit value in the EDX:EAX registers with the destination operand. If they are equal, the 64-bit value in the ECX:EBX register is stored in the destination operand. If the EDX:EAX register and the destination are not equal, the destination is loaded in the EDX:EAX register. The CMPXCHG8B instruction can be combined with the LOCK prefix to perform the operation atomically.

### 30.3.2 Stack Manipulation Instructions

The PUSH, POP, PUSHA (push all registers), and POPA (pop all registers) instructions move data to and from the stack. The PUSH instruction decrements the stack pointer (contained in the ESP register), then copies the source operand to the top of stack (see Figure 30-1). It operates on...
memory operands, immediate operands, and register operands (including segment registers). The
PUSH instruction is commonly used to place parameters on the stack before calling a procedure. It
can also be used to reserve space on the stack for temporary variables.

**Figure 30-1. Operation of the PUSH Instruction**

![Operation of the PUSH Instruction](image)

The PUSH instruction saves the contents of the eight general-purpose registers on the stack (see
Figure 30-2). This instruction simplifies procedure calls by reducing the number of instructions
required to save the contents of the general-purpose registers. The registers are pushed on the stack
in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed,
EBP, ESI, and EDI.

**Figure 30-2. Operation of the PUSHA Instruction**

![Operation of the PUSHA Instruction](image)

The POP instruction copies the word or doubleword at the current top of stack (indicated by the
ESP register) to the location specified with the destination operand, and then increments the ESP
register to point to the new top of stack (see Figure 30-3). The destination operand may specify
a general-purpose register, a segment register, or a memory location.

**Figure 30-3. Operation of the POP Instruction**

![Operation of the POP Instruction](image)
The POPA instruction reverses the effect of the PUSHA instruction. It pops the top eight words or doublewords from the top of the stack into the general-purpose registers, except for the ESP register (see Figure 30-4). If the operand-size attribute is 32, the doublewords on the stack are transferred to the registers in the following order: EDI, ESI, EBP, ignore doubleword, EBX, EDX, ECX, and EAX. The ESP register is restored by the action of popping the stack. If the operand-size attribute is 16, the words on the stack are transferred to the registers in the following order: DI, SI, BP, ignore word, BX, DX, CX, and AX.

Figure 30-4. Operation of the POPA Instruction

![Stack Diagram](image)

30.3.2.1 Type Conversion Instructions

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into quadwords. These instructions are especially useful for converting integers to larger integer formats, because they perform sign extension (see Figure 30-5).

Two kinds of type conversion instructions are provided: simple conversion and move and convert.

Figure 30-5. Sign Extension

![Sign Extension Diagram](image)

30.3.2.2 Simple Conversion

The CBW (convert byte to word), CWDE (convert word to doubleword extended), CWD (convert word to doubleword), and CDQ (convert doubleword to quadword) instructions perform sign extension to double the size of the source operand.

The CBW instruction copies the sign (bit 7) of the byte in the AL register into every bit position of the upper byte of the AX register. The CWDE instruction copies the sign (bit 15) of the word in the AX register into every bit position of the high word of the EAX register.
The CWD instruction copies the sign (bit 15) of the word in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the doubleword in the EAX register into every bit position in the EDX register. The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.

30.3.2.3 Move and Convert

The MOVSX (move with sign extension) and MOVZX (move with zero extension) instructions move the source operand into a register then perform the sign extension.

The MOVSX instruction extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by sign extending the source operand, as shown in Figure 30-5. The MOVZX instruction extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by zero extending the source operand.

30.4 Binary Arithmetic Instructions

The binary arithmetic instructions operate on 8-, 16-, and 32-bit numeric data encoded as signed or unsigned binary integers. Operations include the add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign (negate). The binary arithmetic instructions may also be used in algorithms that operate on decimal (BCD) values.

30.4.1 Addition and Subtraction Instructions

The ADD (add integers), ADC (add integers with carry), SUB (subtract integers), and SBB (subtract integers with borrow) instructions perform addition and subtraction operations on signed or unsigned integer operands.

The ADD instruction computes the sum of two integer operands.

The ADC instruction computes the sum of two integer operands, plus 1 if the CF flag is set. This instruction is used to propagate a carry when adding numbers in stages.

The SUB instruction computes the difference of two integer operands.

The SBB instruction computes the difference of two integer operands, minus 1 if the CF flag is set. This instruction is used to propagate a borrow when subtracting numbers in stages.

30.4.2 Increment and Decrement Instructions

The INC (increment) and DEC (decrement) instructions add 1 to or subtract 1 from an unsigned integer operand, respectively. A primary use of these instructions is for implementing counters.
30.4.3 Comparison and Sign Change Instruction

The CMP (compare) instruction computes the difference between two integer operands and updates the OF, SF, ZF, AF, PF, and CF flags according to the result. The source operands are not modified, nor is the result saved. The CMP instruction is commonly used in conjunction with a Jcc (jump) or SETcc (byte set on condition) instruction, with the latter instructions performing an action based on the result of a CMP instruction.

The NEG (negate) instruction subtracts a signed integer operand from zero. The effect of the NEG instruction is to change the sign of a two's complement operand while keeping its magnitude.

30.4.4 Multiplication and Divide Instructions

The processor provides two multiply instructions, MUL (unsigned multiply) and IMUL (signed multiply), and two divide instructions, DIV (unsigned divide) and IDIV (signed divide).

The MUL instruction multiplies two unsigned integer operands. The result is computed to twice the size of the source operands (for example, if word operands are being multiplied, the result is a doubleword).

The IMUL instruction multiplies two signed integer operands. The result is computed to twice the size of the source operands; however, in some cases the result is truncated to the size of the source operands (see “IMUL—Signed Multiply” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).

The DIV instruction divides one unsigned operand by another unsigned operand and returns a quotient and a remainder.

The IDIV instruction is identical to the DIV instruction, except that IDIV performs a signed division.

30.5 Decimal Arithmetic Instructions

Decimal arithmetic can be performed by combining the binary arithmetic instructions ADD, SUB, MUL, and DIV (discussed in “Binary Arithmetic Instructions”) with the decimal arithmetic instructions. The decimal arithmetic instructions are provided to carry out the following operations:

- To adjust the results of a previous binary arithmetic operation to produce a valid BCD result.
- To adjust the operands of a subsequent binary arithmetic operation so that the operation will produce a valid BCD result.

These instructions operate only on both packed and unpacked BCD values.

30.5.1 Packed BCD Adjustment Instructions

The DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) instructions adjust the results of operations performed on packed BCD integers (see “BCD Integers”). Adding two packed BCD values requires two instructions: an ADD instruction followed by a DAA instruction. The ADD instruction adds (binary addition) the two values and stores the result in the AL register. The DAA instruction then adjusts the value in the AL register to obtain a valid, 2-digit, packed BCD value and sets the CF flag if a decimal carry occurred as the result of the addition.
Likewise, subtracting one packed BCD value from another requires a SUB instruction followed by a DAS instruction. The SUB instruction subtracts (binary subtraction) one BCD value from another and stores the result in the AL register. The DAS instruction then adjusts the value in the AL register to obtain a valid, 2-digit, packed BCD value and sets the CF flag if a decimal borrow occurred as the result of the subtraction.

### 30.5.2 Unpacked BCD Adjustment Instructions

The AAA (ASCII adjust after addition), AAS (ASCII adjust after subtraction), AAM (ASCII adjust after multiplication), and AAD (ASCII adjust before division) instructions adjust the results of arithmetic operations performed in unpacked BCD values (see “BCD Integers”). All these instructions assume that the value to be adjusted is stored in the AL register or, in one instance, the AL and AH registers.

The AAA instruction adjusts the contents of the AL register following the addition of two unpacked BCD values. It converts the binary value in the AL register into a decimal value and stores the result in the AL register in unpacked BCD format (the decimal number is stored in the lower 4 bits of the register and the upper 4 bits are cleared). If a decimal carry occurred as a result of the addition, the CF flag is set and the contents of the AH register are incremented by 1.

The AAS instruction adjusts the contents of the AL register following the subtraction of two unpacked BCD values. Here again, a binary value is converted into an unpacked BCD value. If a borrow was required to complete the decimal subtract, the CF flag is set and the contents of the AH register are decremented by 1.

The AAM instruction adjusts the contents of the AL register following a multiplication of two unpacked BCD values. It converts the binary value in the AL register into a decimal value and stores the least significant digit of the result in the AL register (in unpacked BCD format) and the most significant digit, if there is one, in the AH register (also in unpacked BCD format).

The AAD instruction adjusts a two-digit BCD value so that when the value is divided with the DIV instruction, a valid unpacked BCD result is obtained. The instruction converts the BCD value in registers AH (most significant digit) and AL (least significant digit) into a binary value and stores the result in register AL. When the value in AL is divided by an unpacked BCD value, the quotient and remainder will be automatically encoded in unpacked BCD format.

### 30.6 Logical Instructions

The logical instructions AND, OR, XOR (exclusive or), and NOT perform the standard Boolean operations for which they are named. The AND, OR, and XOR instructions require two operands; the NOT instruction operates on a single operand.

### 30.7 Shift and Rotate Instructions

The shift and rotate instructions rearrange the bits within an operand. These instructions fall into the following classes:

- Shift.
- Double shift.
30.7.1 Shift Instructions

The SAL (shift arithmetic left), SHL (shift logical left), SAR (shift arithmetic right), SHR (shift logical right) instructions perform an arithmetic or logical shift of the bits in a byte, word, or doubleword.

The SAL and SHL instructions perform the same operation (see Figure 30-6). They shift the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand.

Figure 30-6. SHL/SAL Instruction Operation

<table>
<thead>
<tr>
<th>Initial State</th>
<th>CF</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>00 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>After 1-bit SHL/SAL Instruction</td>
<td>1</td>
<td>0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>After 10-bit SHL/SAL Instruction</td>
<td>0</td>
<td>0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

The SHR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 30-7). As with the SHL/SAL instruction, the empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand.

Figure 30-7. SHR Instruction Operation

<table>
<thead>
<tr>
<th>Initial State</th>
<th>CF</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>After 1-bit SHR Instruction</td>
<td>0</td>
<td>0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>After 10-bit SHR Instruction</td>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

The SAR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 30-8). This instruction differs from the SHR instruction in that it preserves the sign of the source operand by clearing empty bit positions if the operand is positive or setting the empty bits if the operand is negative. Again, the CF flag is loaded with the last bit shifted out of the operand.
The SAR and SHR instructions can also be used to perform division by powers of 2 (see “SAL/SAR/SHL/SHR—Shift Instructions” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).

### 30.7.2 Double-Shift Instructions

The SHLD (shift left double) and SHRD (shift right double) instructions shift a specified number of bits from one operand to another (see Figure 30-9). They are provided to facilitate operations on unaligned bit strings. They can also be used to implement a variety of bit string move operations.

#### Figure 30-9. SHLD and SHRD Instruction Operations

The SHLD instruction shifts the bits in the destination operand to the left and fills the empty bit positions (in the destination operand) with bits shifted out of the source operand. The destination and source operands must be the same length (either words or doublewords). The shift count can
range from 0 to 31 bits. The result of this shift operation is stored in the destination operand, and the source operand is not modified. The CF flag is loaded with the last bit shifted out of the destination operand.

The SHRD instruction operates the same as the SHLD instruction except bits are shifted to the left in the destination operand, with the empty bit positions filled with bits shifted out of the source operand.

### 30.7.3 Rotate Instructions

The ROL (rotate left), ROR (rotate right), RCL (rotate through carry left) and RCR (rotate through carry right) instructions rotate the bits in the destination operand out of one end and back through the other end (see Figure 30-10). Unlike a shift, no bits are lost during a rotation. The rotate count can range from 0 to 31.

**Figure 30-10. ROL, ROR, RCL, and RCR Instruction Operations**

The ROL instruction rotates the bits in the operand to the left (toward more significant bit locations). The ROR instruction rotates the operand right (toward less significant bit locations).

The RCL instruction rotates the bits in the operand to the left, through the CF flag. This instruction treats the CF flag as a one-bit extension on the upper end of the operand. Each bit which exits from the most significant bit location of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the least significant bit location of the operand.

The RCR instruction rotates the bits in the operand to the right through the CF flag.

For all the rotate instructions, the CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand. The value of this flag can then be tested by a conditional jump instruction (JC or JNC).
30.8 Bit And Byte Instructions

The bit and byte instructions operate on bit or byte strings. They are divided into four groups:

- Bit test and modify instructions.
- Bit scan instructions.
- Byte set on condition.
- Test.

30.8.1 Bit Test and Modify Instructions

The bit test and modify instructions (see Table 30-3) operate on a single bit, which can be in an operand. The location of the bit is specified as an offset from the least significant bit of the operand. When the processor identifies the bit to be tested and modified, it first loads the CF flag with the current value of the bit. Then it assigns a new value to the selected bit, as determined by the modify operation for the instruction.

Table 30-3. Bit Test and Modify Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect on CF Flag</th>
<th>Effect on Selected Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT (Bit Test)</td>
<td>CF flag ← Selected Bit</td>
<td>No effect</td>
</tr>
<tr>
<td>BTS (Bit Test and Set)</td>
<td>CF flag ← Selected Bit</td>
<td>Selected Bit ← 1</td>
</tr>
<tr>
<td>BTR (Bit Test and Reset)</td>
<td>CF flag ← Selected Bit</td>
<td>Selected Bit ← 0</td>
</tr>
<tr>
<td>BTC (Bit Test and Complement)</td>
<td>CF flag ← Selected Bit</td>
<td>Selected Bit ← NOT (Selected Bit)</td>
</tr>
</tbody>
</table>

30.8.2 Bit Scan Instructions

The BSF (bit scan forward) and BSR (bit scan reverse) instructions scan a bit string in a source operand for a set bit and store the bit index of the first set bit found in a destination register. The bit index is the offset from the least significant bit (bit 0) in the bit string to the first set bit. The BSF instruction scans the source operand low-to-high (from bit 0 of the source operand toward the most significant bit); the BSR instruction scans high-to-low (from the most significant bit toward the least significant bit).

30.8.3 Byte Set On Condition Instructions

The SETcc (set byte on condition) instructions set a destination-operand byte to 0 or 1, depending on the state of selected status flags (CF, OF, SF, ZF, and PF) in the EFLAGS register. The suffix (cc) added to the SET mnemonic determines the condition being tested for. For example, the SETO instruction tests for overflow. If the OF flag is set, the destination byte is set to 1; if OF is clear, the destination byte is cleared to 0. “EFLAGS Condition Codes” lists the conditions it is possible to test for with this instruction.
30.8.4 Test Instruction

The TEST instruction performs a logical AND of two operands and sets the SF, ZF, and PF flags according to the results. The flags can then be tested by the conditional jump or loop instructions or the SETcc instructions. The TEST instruction differs from the AND instruction in that it does not alter either of the operands.

30.9 Control Transfer Instructions

The processor provides both conditional and unconditional control transfer instructions to direct the flow of program execution. Conditional transfers are taken only for specified states of the status flags in the EFLAGS register. Unconditional control transfers are always executed.

30.9.1 Unconditional Transfer Instructions

The JMP, CALL, RET, INT, and IRET instructions transfer program control to another location (destination address) in the instruction stream. The destination can be within the same code segment (near transfer) or in a different code segment (far transfer).

30.9.1.1 Jump Instruction

The JMP (jump) instruction unconditionally transfers program control to a destination instruction. The transfer is one-way; that is, a return address is not saved. A destination operand specifies the address (the instruction pointer) of the destination instruction. The address can be a relative address or an absolute address.

A relative address is a displacement (offset) with respect to the address in the EIP register. The destination address (a near pointer) is formed by adding the displacement to the address in the EIP register. The displacement is specified with a signed integer, allowing jumps either forward or backward in the instruction stream.

An absolute address is a offset from address 0 of a segment. It can be specified in either of the following ways:

- An address in a general-purpose register. This address is treated as a near pointer, which is copied into the EIP register. Program execution then continues at the new address within the current code segment.

- An address specified using the standard addressing modes of the processor. Here, the address can be a near pointer or a far pointer. If the address is for a near pointer, the address is translated into an offset and copied into the EIP register. If the address is for a far pointer, the address is translated into a segment selector (which is copied into the CS register) and an offset (which is copied into the EIP register).

In protected mode, the JMP instruction also allows jumps to a call gate, a task gate, and a task-state segment.

30.9.1.2 Call and Return Instructions

The CALL (call procedure) and RET (return from procedure) instructions allow a jump from one procedure (or subroutine) to another and a subsequent jump back (return) to the calling procedure.
The CALL instruction transfers program control from the current (or calling procedure) to another procedure (the called procedure). To allow a subsequent return to the calling procedure, the CALL instruction saves the current contents of the EIP register on the stack before jumping to the called procedure. The EIP register (prior to transferring program control) contains the address of the instruction following the CALL instruction. When this address is pushed on the stack, it is referred to as the return instruction pointer or return address.

The address of the called procedure (the address of the first instruction in the procedure being jumped to) is specified in a CALL instruction the same way as it is in a JMP instruction (see “Jump Instruction”). The address can be specified as a relative address or an absolute address. If an absolute address is specified, it can be either a near or a far pointer.

The RET instruction transfers program control from the procedure currently being executed (the called procedure) back to the procedure that called it (the calling procedure). Transfer of control is accomplished by copying the return instruction pointer from the stack into the EIP register. Program execution then continues with the instruction pointed to by the EIP register.

The RET instruction has an optional operand, the value of which is added to the contents of the ESP register as part of the return operation. This operand allows the stack pointer to be incremented to remove parameters from the stack that were pushed on the stack by the calling procedure.

See “Calling Procedures Using CALL and RET”, for more information on the mechanics of making procedure calls with the CALL and RET instructions.

### 30.9.1.3 Return From Interrupt Instruction

When the processor services an interrupt, it performs an implicit call to an interrupt-handling procedure. The IRET (return from interrupt) instruction returns program control from an interrupt handler to the interrupted procedure (that is, the procedure that was executing when the interrupt occurred). The IRET instruction performs a similar operation to the RET instruction (see “Call and Return Instructions”) except that it also restores the EFLAGS register from the stack. The contents of the EFLAGS register are automatically stored on the stack along with the return instruction pointer when the processor services an interrupt.

### 30.9.2 Conditional Transfer Instructions

The conditional transfer instructions execute jumps or loops that transfer program control to another instruction in the instruction stream if specified conditions are met. The conditions for control transfer are specified with a set of condition codes that define various states of the status flags (CF, ZF, OF, PF, and SF) in the EFLAGS register.

### 30.9.2.1 Conditional Jump Instructions

The Jcc (conditional) jump instructions transfer program control to a destination instruction if the conditions specified with the condition code (cc) associated with the instruction are satisfied (see Table 30-4). If the condition is not satisfied, execution continues with the instruction following the Jcc instruction. As with the JMP instruction, the transfer is one-way; that is, a return address is not saved.
Table 30-4. Conditional Jump Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Condition (Flag States)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned Conditional Jumps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JA/JNBE</td>
<td>(CF or ZF)=0</td>
<td>Above/not below or equal</td>
</tr>
<tr>
<td>JAE/JNB</td>
<td>CF=0</td>
<td>Above or equal/not below</td>
</tr>
<tr>
<td>JB/JNAE</td>
<td>CF=1</td>
<td>Below/not above or equal</td>
</tr>
<tr>
<td>JBE/JNA</td>
<td>(CF or ZF)=1</td>
<td>Below or equal/not above</td>
</tr>
<tr>
<td>JC</td>
<td>CF=1</td>
<td>Carry</td>
</tr>
<tr>
<td>JE/JZ</td>
<td>ZF=1</td>
<td>Equal/zero</td>
</tr>
<tr>
<td>JNC</td>
<td>CF=0</td>
<td>Not carry</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>ZF=0</td>
<td>Not equal/not zero</td>
</tr>
<tr>
<td>JNP/JPO</td>
<td>PF=0</td>
<td>Not parity/parity odd</td>
</tr>
<tr>
<td>JP/JPE</td>
<td>PF=1</td>
<td>Parity/parity even</td>
</tr>
<tr>
<td>JCXZ</td>
<td>CX=0</td>
<td>Register CX is zero</td>
</tr>
<tr>
<td>JECXZ</td>
<td>ECX=0</td>
<td>Register ECX is zero</td>
</tr>
<tr>
<td><strong>Signed Conditional Jumps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JG/JNLE</td>
<td>((SF xor OF) or ZF) =0</td>
<td>Greater/not less or equal</td>
</tr>
<tr>
<td>JGE/JNL</td>
<td>(SF xor OF)=0</td>
<td>Greater or equal/not less</td>
</tr>
<tr>
<td>JL/JNGE</td>
<td>(SF xor OF)=1</td>
<td>Less/not greater or equal</td>
</tr>
<tr>
<td>JLE/JNG</td>
<td>((SF xor OF) or ZF)=1</td>
<td>Less or equal/not greater</td>
</tr>
<tr>
<td>JNO</td>
<td>OF=0</td>
<td>Not overflow</td>
</tr>
<tr>
<td>JNS</td>
<td>SF=0</td>
<td>Not sign (non-negative)</td>
</tr>
<tr>
<td>JO</td>
<td>OF=1</td>
<td>Overflow</td>
</tr>
<tr>
<td>JS</td>
<td>SF=1</td>
<td>Sign (negative)</td>
</tr>
</tbody>
</table>

The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The Jcc instructions do not support far transfers; however, far transfers can be accomplished with a combination of a Jcc and a JMP instruction (see “Jcc—Jump if Condition Is Met” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 2*).

Table 30-4 shows the mnemonics for the Jcc instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter “J” to form the mnemonic for a Jcc instruction. The instructions are divided into two groups: unsigned and signed conditional jumps. These groups correspond to the results of operations performed on unsigned and signed integers, respectively. Those instructions listed as pairs (for example, JA/JNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The JCXZ and JECXZ instructions test the CX and ECX registers, respectively, instead of one or more status flags. See “Jump If Zero Instructions” for more information about these instructions.
30.9.2.2 Loop Instructions

The LOOP, LOOPE (loop while equal), LOOPZ (loop while zero), LOOPNE (loop while not equal), and LOOPNZ (loop while not zero) instructions are conditional jump instructions that use the value of the ECX register as a count for the number of times to execute a loop. All the loop instructions decrement the count in the ECX register each time they are executed and terminate a loop when zero is reached. The LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions also accept the ZF flag as a condition for terminating the loop before the count reaches zero.

The LOOP instruction decrements the contents of the ECX register (or the CX register, if the address-size attribute is 16), then tests the register for the loop-termination condition. If the count in the ECX register is non-zero, program control is transferred to the instruction address specified by the destination operand. The destination operand is a relative address (that is, an offset relative to the contents of the EIP register), and it generally points to the first instruction in the block of code that is to be executed in the loop. When the count in the ECX register reaches zero, program control is transferred to the instruction immediately following the LOOP instruction, which terminates the loop. If the count in the ECX register is zero when the LOOP instruction is first executed, the register is pre-decremented to FFFFFFFFH, causing the loop to be executed $2^{32}$ times.

The LOOPE and LOOPZ instructions perform the same operation (they are mnemonics for the same instruction). These instructions operate the same as the LOOP instruction, except that they also test the ZF flag. If the count in the ECX register is not zero and the ZF flag is set, program control is transferred to the destination operand. When the count reaches zero or the ZF flag is clear, the loop is terminated by transferring program control to the instruction immediately following the LOOPE/LOOPZ instruction.

The LOOPNE and LOOPNZ instructions (mnemonics for the same instruction) operate the same as the LOOPE/LOOPPZ instructions, except that they terminate the loop if the ZF flag is set.

30.9.2.3 Jump If Zero Instructions

The JECXZ (jump if ECX zero) instruction jumps to the location specified in the destination operand if the ECX register contains the value zero. This instruction can be used in combination with a loop instruction (LOOP, LOOPE, LOOPZ, LOOPNE, or LOOPNZ) to test the ECX register prior to beginning a loop. As described in “Loop Instructions”, the loop instructions decrement the contents of the ECX register before testing for zero. If the value in the ECX register is zero initially, it will be decremented to FFFFFFFFH on the first loop instruction, causing the loop to be executed $2^{32}$ times. To prevent this problem, a JECXZ instruction can be inserted at the beginning of the code block for the loop, causing a jump out the loop if the EAX register count is initially zero. When used with repeated string scan and compare instructions, the JECXZ instruction can determine whether the loop terminated because the count reached zero or because the scan or compare conditions were satisfied.

The JCXZ (jump if CX is zero) instruction operates the same as the JECXZ instruction when the 16-bit address-size attribute is used. Here, the CX register is tested for zero.

30.9.3 Software Interrupts

The INT n (software interrupt), INTO (interrupt on overflow), and BOUND (detect value out of range) instructions allow a program to explicitly raise a specified interrupt or exception, which in turn causes the handler routine for the interrupt or exception to be called.
The INT \( n \) instruction can raise any of the processor’s interrupts or exceptions by encoding the vector number or the interrupt or exception in the instruction. This instruction can be used to support software generated interrupts or to test the operation of interrupt and exception handlers. The IRET instruction (see “Return From Interrupt Instruction”) allows returns from interrupt handling routines.

The INTO instruction raises the overflow exception, if the OF flag is set. If the flag is clear, execution continues without raising the exception. This instruction allows software to access the overflow exception handler explicitly to check for overflow conditions.

The BOUND instruction compares a signed value against upper and lower bounds, and raises the “BOUND range exceeded” exception if the value is less than the lower bound or greater than the upper bound. This instruction is useful for operations such as checking an array index to make sure it falls within the range defined for the array.

### 30.10 String Operations

The MOVBS (Move String), CMPS (Compare string), SCAS (Scan string), LODS (Load string), and STOS (Store string) instructions permit large data structures, such as alphanumeric character strings, to be moved and examined in memory. These instructions operate on individual elements in a string, which can be a byte, word, or doubleword. The string elements to be operated on are identified with the ESI (source string element) and EDI (destination string element) registers. Both of these registers contain absolute addresses (offsets into a segment) that point to a string element.

By default, the ESI register addresses the segment identified with the DS segment register. A segment-override prefix allows the ESI register to be associated with the CS, SS, ES, FS, or GS segment register. The EDI register addresses the segment identified with the ES segment register; no segment override is allowed for the EDI register. The use of two different segment registers in the string instructions permits operations to be performed on strings located in different segments. Or by associating the ESI register with the ES segment register, both the source and destination strings can be located in the same segment. (This latter condition can also be achieved by loading the DS and ES segment registers with the same segment selector and allowing the ESI register to default to the DS register.)

The MOVBS instruction moves the string element addressed by the ESI register to the location addressed by the EDI register. The assembler recognizes three “short forms” of this instruction, which specify the size of the string to be moved: MOVBS (move byte string), MOVSW (move word string), and MOVSD (move doubleword string).

The CMPS instruction subtracts the destination string element from the source string element and updates the status flags (CF, ZF, OF, SF, PF, and AF) in the EFLAGS register according to the results. Neither string element is written back to memory. The assembler recognizes three “short forms” of the CMPS instruction: CMPSB (compare byte strings), CMPSW (compare word strings), and CMPSD (compare doubleword strings).

The SCAS instruction subtracts the destination string element from the contents of the EAX, AX, or AL register (depending on operand length) and updates the status flags according to the results. The string element and register contents are not modified. The following “short forms” of the SCAS instruction specifies the operand length: SCASB (scan byte string), SCASW (scan word string), and SCASD (scan doubleword string).
The LODS instruction loads the source string element identified by the ESI register into the EAX register (for a doubleword string), the AX register (for a word string), or the AL register (for a byte string). The “short forms” for this instruction are LODSB (load byte string), LODSW (load word string), and LODSD (load doubleword string). This instruction is usually used in a loop, where other instructions process each element of the string after they are loaded into the target register.

The STOS instruction stores the source string element from the EAX (doubleword string), AX (word string), or AL (byte string) register into the memory location identified with the EDI register. The “short forms” for this instruction are STOSB (store byte string), STOSW (store word string), and STOSD (store doubleword string). This instruction is also normally used in a loop. Here a string is commonly loaded into the register with a LODS instruction, operated on by other instructions, and then stored again in memory with a STOS instruction.

The I/O instructions (see “I/O Instructions”) also perform operations on strings in memory.

30.10.1 Repeating String Operations

The string instructions described in “String Operations” perform one iteration of a string operation. To operate strings longer than a doubleword, the string instructions can be combined with a repeat prefix (REP) to create a repeating instruction or be placed in a loop.

When used in string instructions, the ESI and EDI registers are automatically incremented or decremented after each iteration of an instruction to point to the next element (byte, word, or doubleword) in the string. String operations can thus begin at higher addresses and work toward lower ones, or they can begin at lower addresses and work toward higher ones. The DF flag in the EFLAGS register controls whether the registers are incremented (DF=0) or decremented (DF=1). The STD and CLD instructions set and clear this flag, respectively.

The following repeat prefixes can be used in conjunction with a count in the ECX register to cause a string instruction to repeat:

- REP—Repeat while the ECX register not zero.
- REPE/REPZ—Repeat while the ECX register not zero and the ZF flag is set.
- REPNE/REPNZ—Repeat while the ECX register not zero and the ZF flag is clear.

When a string instruction has a repeat prefix, the operation executes until one of the termination conditions specified by the prefix is satisfied. The REPE/REPZ and REPNE/REPNZ prefixes are used only with the CMPS and SCAS instructions. Also, note that a REP STOS instruction is the fastest way to initialize a large block of memory.

30.11 I/O Instructions

The IN (input from port to register), INS (input from port to string), OUT (output from register to port), and OUTS (output string to port) instructions move data between the processor’s I/O ports and either a register or memory.

The register I/O instructions (IN and OUT) move data between an I/O port and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The I/O port being read or written to is specified with an immediate operand or an address in the DX register.
The block I/O instructions (INS and OUTS) instructions move blocks of data (strings) between an I/O port and memory. These instructions operate similar to the string instructions (see “String Operations”). The ESI and EDI registers are used to specify string elements in memory and the repeat prefixes (REP) are used to repeat the instructions to implement block moves. The assembler recognizes the following alternate mnemonics for these instructions: INSB (input byte), INSW (input word), and INSD (input doubleword), and OUTB (output byte), OUTW (output word), and OUTD (output doubleword).

The INS and OUTS instructions use an address in the DX register to specify the I/O port to be read or written to.

### 30.12 Enter and Leave Instructions

The ENTER and LEAVE instructions provide machine-language support for procedure calls in block-structured languages, such as C and Pascal. These instructions and the call and return mechanism that they support are described in detail in “Procedure Calls for Block-Structured Languages”.

### 30.13 EFLAGS Instructions

The EFLAGS instructions allow the state of selected flags in the EFLAGS register to be read or modified.

#### 30.13.1 Carry and Direction Flag Instructions

The STC (set carry flag), CLC (clear carry flag), and CMC (complement carry flag) instructions allow the CF flags in the EFLAGS register to be modified directly. They are typically used to initialize the CF flag to a known state before an instruction that uses the flag in an operation is executed. They are also used in conjunction with the rotate-with-carry instructions (RCL and RCR).

The STD (set direction flag) and CLD (clear direction flag) instructions allow the DF flag in the EFLAGS register to be modified directly. The DF flag determines the direction in which index registers ESI and EDI are stepped when executing string processing instructions. If the DF flag is clear, the index registers are incremented after each iteration of a string instruction; if the DF flag is set, the registers are decremented.

#### 30.13.2 Interrupt Flag Instructions

The STI (set interrupt flag) and CTI (clear interrupt flag) instructions allow the interrupt IF flag in the EFLAGS register to be modified directly. The IF flag controls the servicing of hardware-generated interrupts (those received at the processor’s INTR pin). If the IF flag is set, the processor services hardware interrupts; if the IF flag is clear, hardware interrupts are masked.

#### 30.13.3 EFLAGS Transfer Instructions

The EFLAGS transfer instructions allow groups of flags in the EFLAGS register to be copied to a register or memory or be loaded from a register or memory.
The LAHF (load AH from flags) and SAHF (store AH into flags) instructions operate on five of the EFLAGS status flags (SF, ZF, AF, PF, and CF). The LAHF instruction copies the status flags to bits 7, 6, 4, 2, and 0 of the AH register, respectively. The contents of the remaining bits in the register (bits 5, 3, and 1) are undefined, and the contents of the EFLAGS register remain unchanged. The SAHF instruction copies bits 7, 6, 4, 2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags, respectively in the EFLAGS register.

The PUSHF (push flags), PUSHFD (push flags double), POPF (pop flags), and POPFD (pop flags double) instructions copy the flags in the EFLAGS register to and from the stack. The PUSHF instruction pushes the lower word of the EFLAGS register onto the stack (see Figure 30-11). The PUSHFD instruction pushes the entire EFLAGS register onto the stack (with the RF and VM flags read as clear).

The POPF instruction pops a word from the stack into the EFLAGS register. Only bits 11, 10, 8, 7, 6, 4, 2, and 0 of the EFLAGS register are affected with all uses of this instruction. If the current privilege level (CPL) of the current code segment is 0 (most privileged), the IOPL bits (bits 13 and 12) also are affected. If the I/O privilege level (IOPL) is greater than or equal to the CPL, numerically, the IF flag (bit 9) also is affected.

The POPFD instruction pops a doubleword into the EFLAGS register. This instruction can change the state of the AC bit (bit 18) and the ID bit (bit 21), as well as the bits affected by a POPF instruction. The restrictions for changing the IOPL bits and the IF flag that were given for the POPF instruction also apply to the POPFD instruction.

### 30.13.4 Interrupt Flag Instructions

The CLI (clear interrupt flag) and STI (set interrupt flag) instructions clear and set the interrupt flag (IF) in the EFLAGS register, respectively. Clearing the IF flag causes external interrupts to be ignored. The ability to execute these instructions depends on the operating mode of the processor and the current privilege level (CPL) of the program or task attempting to execute these instructions.

### 30.14 segment register instructions

The processor provides a variety of instructions that address the segment registers of the processor directly. These instructions are only used when an operating system or executive is using the segmented or the real-address mode memory model.
30.14.1 Segment-Register Load and Store Instructions

The MOV instruction (introduced in “General-Purpose Data Movement Instructions”) and the PUSH and POP instructions (introduced in “Stack Manipulation Instructions”) can transfer 16-bit segment selectors to and from segment registers (DS, ES, FS, GS, and SS). The transfers are always made to or from a segment register and a general-purpose register or memory. Transfers between segment registers are not supported.

The POP and MOV instructions cannot place a value in the CS register. Only the far control-transfer versions of the JMP, CALL, and RET instructions (see “Far Control Transfer Instructions”) affect the CS register directly.

30.14.2 Far Control Transfer Instructions

The JMP and CALL instructions (see “Control Transfer Instructions”) both accept a far pointer as a source operand to transfer program control to a segment other than the segment currently being pointed to by the CS register. When a far call is made with the CALL instruction, the current values of the EIP and CS registers are both pushed on the stack.

The RET instruction (see “Call and Return Instructions”) can be used to execute a far return. Here, program control is transferred from a code segment that contains a called procedure back to the code segment that contained the calling procedure. The RET instruction restores the values of the CS and EIP registers for the calling procedure from the stack.

30.14.3 Software Interrupt Instructions

The software interrupt instructions INT, INTO, BOUND, and IRET (see “Software Interrupts”) can also call and return from interrupt and exception handler procedures that are located in a code segment other than the current code segment. With these instructions, however, the switching of code segments is handled transparently from the application program.

30.14.4 Load Far Pointer Instructions

The load far pointer instructions LDS (load far pointer using DS), LES (load far pointer using ES), LFS (load far pointer using FS), LGS (load far pointer using GS), and LSS (load far pointer using SS) load a far pointer from memory into a segment register and a general-purpose general register. The segment selector part of the far pointer is loaded into the selected segment register and the offset is loaded into the selected general-purpose register.

30.15 Miscellaneous Instructions

The following instructions perform miscellaneous operations that are of interest to applications programmers.
30.15.1 Address Computation Instruction

The LEA (load effective address) instruction computes the effective address in memory (offset within a segment) of a source operand and places it in a general-purpose register. This instruction can interpret any of the Pentium Pro processor’s addressing modes and can perform any indexing or scaling that may be needed. It is especially useful for initializing the ESI or EDI registers before the execution of string instructions or for initializing the EBX register before an XLAT instruction.

30.15.2 Table Lookup Instructions

The XLAT and XLATB (table lookup) instructions replace the contents of the AL register with a byte read from a translation table in memory. The initial value in the AL register is interpreted as an unsigned index into the translation table. This index is added to the contents of the EBX register (which contains the base address of the table) to calculate the address of the table entry. These instructions are used for applications such as converting character codes from one alphabet into another (for example, an ASCII code could be used to look up its EBCDIC equivalent in a table).

30.15.3 Processor Identification Instruction

The CPUID (processor identification) instruction provides information about the processor on which the instruction is executed. To obtain processor information, a value of from 0 to 2 is loaded in the EAX register and then the CPUID instruction is executed. The resulting processor information is placed in the EAX, EBX, ECX, and EDX registers. Table 30-5 shows the information that is provided depending on the value initially entered in the EAX register. See “Processor Identification”, for detailed information on the output of the CPUID instruction.

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Maximum CPUID input value.</td>
</tr>
<tr>
<td></td>
<td>Vendor identification string (“GenuineIntel”).</td>
</tr>
<tr>
<td>1</td>
<td>Version information (family ID, model ID, and stepping ID).</td>
</tr>
<tr>
<td></td>
<td>Feature information (identifies the feature set for the processor model).</td>
</tr>
<tr>
<td>2</td>
<td>Cache information (about the processor’s internal cache memory).</td>
</tr>
</tbody>
</table>

30.15.4 No-Operation and Undefined Instructions

The NOP (no operation) instruction increments the EIP register to point at the next instruction, but affects nothing else.

The UD2 (undefined) instruction generates an invalid opcode exception. Intel reserves the opcode for this instruction for this function. The instruction is provided to allow software to test an invalid opcode exception handler.