x86 Assembly Language III

CMSC 313
Sections 01, 02

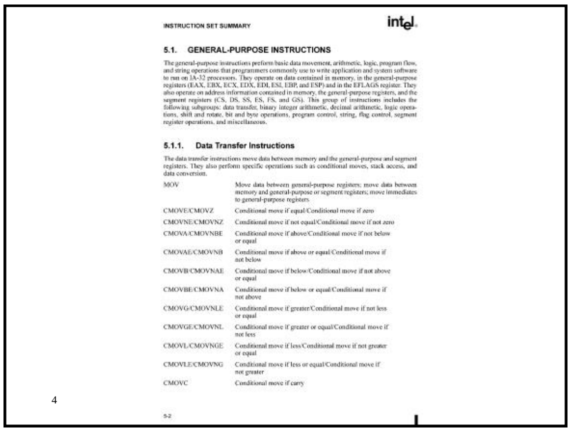
i386 Instruction Overview

2

i386 Instruction Set Overview

- General Purpose Instructions
  - works with data in the general purpose registers
- Floating Point Instructions
  - floating point arithmetic
  - data stored in separate floating point registers
- Single Instruction Multiple Data (SIMD) Extensions
  - MMX, SSE, SSE2
- System Instructions
  - Sets up control registers at boot time

3 UMBC, CMSC313, Richard Chang <chang@umbc.edu>



5.1. GENERAL-PURPOSE INSTRUCTIONS

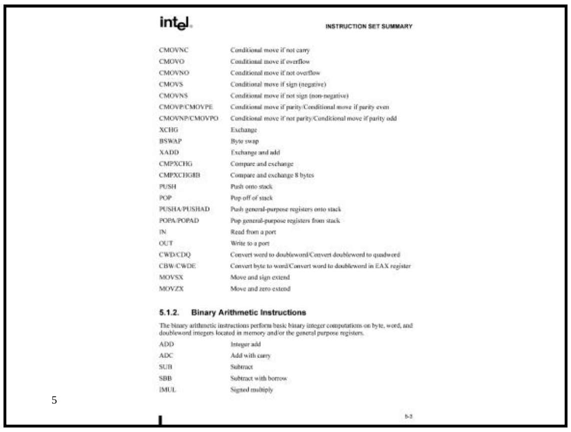
The general-purpose instructions preform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run on IA-32 processors. They operate on data contained in memory, in the general-purpose registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP) and in the EFLAGS register. They also operate on address information contained in memory, the general-purpose registers, and the segment registers (CS, DS, SS, ES, FS, and GS). This group of instructions includes the following subgroups: data transfer, binary integer arithmetic, decimal arithmetic, logic operations, shift and rotate, bit and byte operations, program control, string, flag control, segment register operations, and miscellaneous.

5.1.1. Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater
CMOVC	Conditional move if carry

4



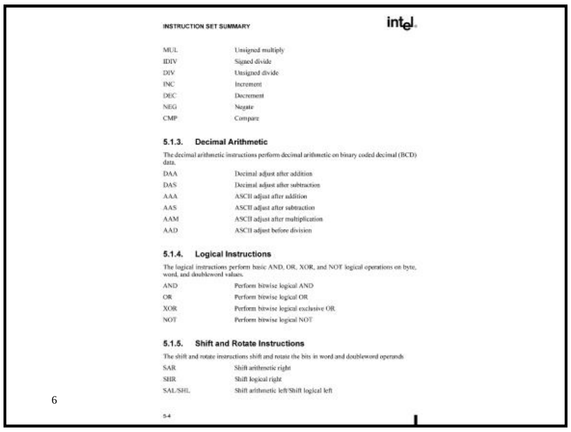
CMOVNC	Conditional move if not carry
CMOVO	Conditional move if overflow
CMOVNO	Conditional move if not overflow
CMOVS	Conditional move if sign (negative)
CMOVNS	Conditional move if not sign (non-negative)
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd
XCHG	Exchange
BSWAP	Byte swap
XADD	Exchange and add
CMPXCHG	Compare and exchange
CMPXCHG8B	Compare and exchange 8 bytes
PUSH	Push onto stack
POP	Pop off of stack
PUSHA/PUSHAD	Push general-purpose registers onto stack
POPA/POPAD	Pop general-purpose registers from stack
IN	Read from a port
OUT	Write to a port
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register
MOVSX	Move and sign extend
MOVZX	Move and zero extend

5.1.2. Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADD	Integer add
ADC	Add with carry
SUB	Subtract
SBB	Subtract with borrow
IMUL	Signed multiply

5



MUL	Unsigned multiply
IDIV	Signed divide
DIV	Unsigned divide
INC	Increment
DEC	Decrement
NEG	Negate
CMP	Compare

5.1.3. Decimal Arithmetic

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

DAA	Decimal adjust after addition
DAS	Decimal adjust after subtraction
AAA	ASCII adjust after addition
AAS	ASCII adjust after subtraction
AAM	ASCII adjust after multiplication
AAD	ASCII adjust before division

5.1.4. Logical Instructions

The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.

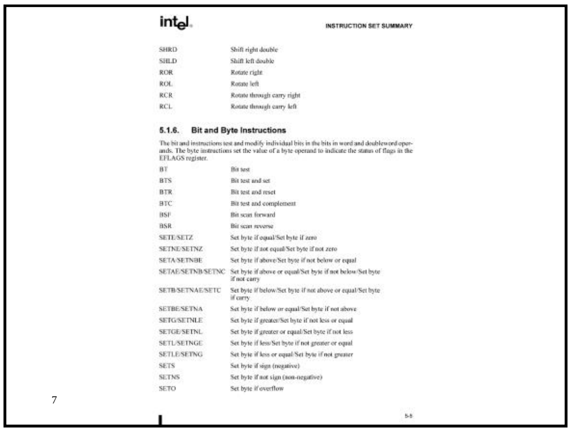
AND	Perform bitwise logical AND
OR	Perform bitwise logical OR
XOR	Perform bitwise logical exclusive OR
NOT	Perform bitwise logical NOT

5.1.5. Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in word and doubleword operands

SAR	Shift arithmetic right
SHR	Shift logical right
SAL/SHL	Shift arithmetic left/Shift logical left

6



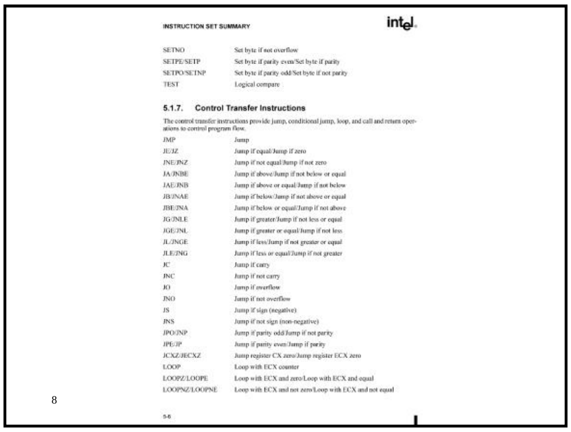
SHRD	Shift right double
SHLD	Shift left double
ROR	Rotate right
ROL	Rotate left
RCR	Rotate through carry right
RCL	Rotate through carry left

5.1.6. Bit and Byte Instructions

The bit and instructions test and modify individual bits in the bits in word and doubleword operands. The byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.

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

7



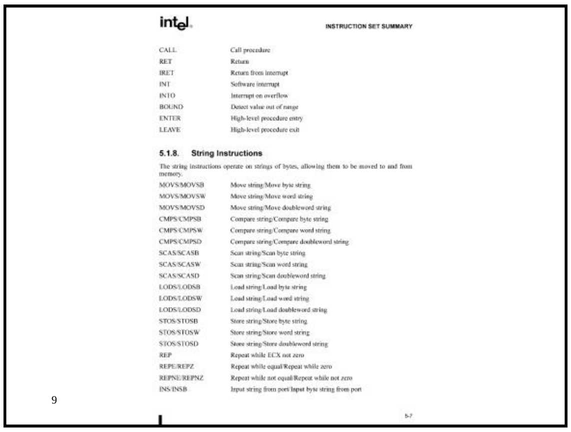
SETNO	Set byte if not overflow
SETPE/SETP	Set byte if parity even/Set byte if parity
SETPO/SETNP	Set byte if parity odd/Set byte if not parity
TEST	Logical compare

5.1.7. Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

JMP	Jump
JE/JZ	Jump if equal/Jump if zero
JNE/JNZ	Jump if not equal/Jump if not zero
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater
JC	Jump if carry
JNC	Jump if not carry
JO	Jump if overflow
JNO	Jump if not overflow
JS	Jump if sign (negative)
JNS	Jump if not sign (non-negative)
JPO/JNP	Jump if parity odd/Jump if not parity
JPE/JP	Jump if parity even/Jump if parity
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero
LOOP	Loop with ECX counter
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal

8



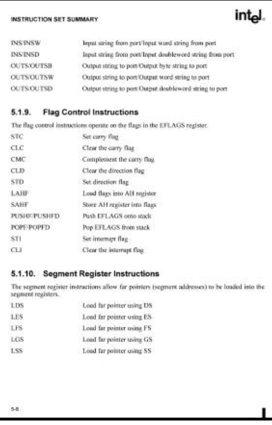
CALL	Call procedure
RET	Return
IRET	Return from interrupt
INT	Software interrupt
INTO	Interrupt on overflow
BOUND	Detect value out of range
ENTER	High-level procedure entry
LEAVE	High-level procedure exit

5.1.8. String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

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

9



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

5.1.9. Flag Control Instructions

The flag control instructions operate on the flags in the EFLAGS register.

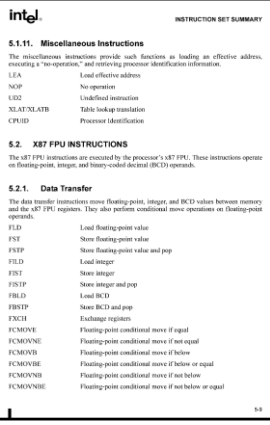
STC	Set carry flag
CLC	Clear the carry flag
CMC	Complement the carry flag
CLD	Clear the direction flag
STD	Set direction flag
LAHF	Load flags into AH register
SAHF	Store AH register into flags
PUSHF/PUSHFD	Push EFLAGS onto stack
POPF/POPFD	Pop EFLAGS from stack
STI	Set interrupt flag
CLI	Clear the interrupt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

5.1.10. Segment Register Instructions

The segment register instructions allow far pointers (segment addresses) to be loaded into the segment registers.

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

10



5.1.11. Miscellaneous Instructions

The miscellaneous instructions provide such functions as loading an effective address, executing a "no-operation," and retrieving processor identification information.

LEA	Load effective address
NOP	No operation
UD2	Undefined instruction
XLAT/XLATB	Table lookup translation
CPUID	Processor Identification

5.2. X87 FPU INSTRUCTIONS

The x87 FPU instructions are executed by the processor's x87 FPU. These instructions operate on floating-point, integer, and binary-coded decimal (BCD) operands.

5.2.1. Data Transfer

The data transfer instructions move floating-point, integer, and BCD values between memory and the x87 FPU registers. They also perform conditional move operations on floating-point operands.

FLD	Load floating-point value
FST	Store floating-point value
FSTP	Store floating-point value 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

11

Common Instructions

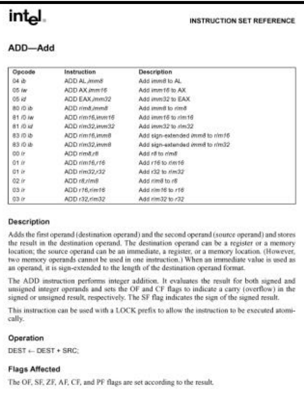
- Basic Instructions
  - ADD, SUB, INC, DEC, MOV, NOP
- Branching Instructions
  - JMP, CMP, Jcc
- More Arithmetic Instructions
  - NEG, MUL, IMUL, DIV, IDIV
- Logical (bit manipulation) Instructions
  - AND, OR, NOT, SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR
- Subroutine Instructions
  - PUSH, POP, CALL, RET

12

READ THE FRIENDLY MANUAL (RTFM)

- **Best Source: Intel Instruction Set Reference**
  - Available off the course web page in PDF
  - Download it, you'll need it
- **Other sources:**
  - Appendix A of *Assembly Language Step-by-Step*
- **Questions to ask:**
  - Basic function? (e.g., adds two numbers)
  - Addressing modes supported? (e.g., register to register)
  - Side effects? (e.g., OF modified)

13 UMBC, CMSC313, Richard Chang <chang@umbc.edu>



INSTRUCTION SET REFERENCE

ADD—Add

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

Description

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

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST ← DEST + SRC;
```

Flags Affected

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

14

Intel Manual's Addressing Mode Notation

- ***r8***: **One of the 8-bit registers AL, CL, DL, BL, AH, CH, DH, or BH.**
- ***r16***: **One of the 16-bit registers AX, CX, DX, BX, SP, BP, SI, or DI.**
- ***r32***: **One of the 32-bit registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.**
- ***imm8***: **An immediate 8-bit value.**
- ***imm16***: **An immediate 16-bit value.**
- ***imm32***: **An immediate 32-bit value.**
- ***r/m8***: **An 8-bit operand that is either the contents of an 8-bit register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.**
- ***r/m16***: **A 16-bit register (AX, BX, CX, DX, SP, BP, SI, and DI) or memory operand used for instructions whose operand-size attribute is 16 bits.**
- ***r/m32***: **A 32-bit register (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI) or memory operand used for instructions whose operand-size attribute is 32 bits.**

15

The EFLAGS Register

- A special 32-bit register that contains “results” of previous instructions
  - OF = overflow flag, indicates two’s complement overflow.
  - SF = sign flag, indicates a negative result.
  - ZF = zero flag, indicates the result was zero.
  - CF = carry flag, indicates unsigned overflow, also used in shifting
- An operation may set, clear, modify or test a flag.
- Some operations leave a flag undefined.

BASIC EXECUTION ENVIRONMENT

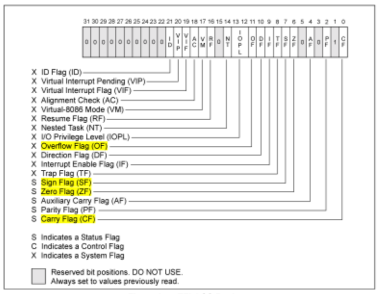
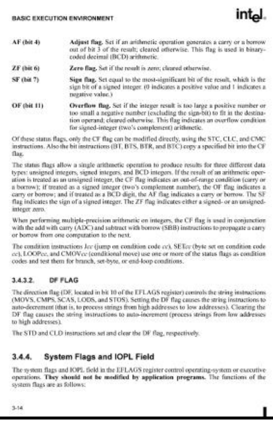

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
X Overflow Flag (OF)
X Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

Figure 3-7. EFLAGS Register



BASIC EXECUTION ENVIRONMENT

AF (bit 4)	**Adjust flag.** Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
ZF (bit 6)	**Zero flag.** Set if the result is zero; cleared otherwise.
SF (bit 7)	**Sign flag.** Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
OF (bit 11)	**Overflow flag.** Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions J*cc* (jump on condition code *cc*), SET*cc* (byte set on condition code *cc*), LOOP*cc*, and CMOV*cc* (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

3.4.3.2. DF FLAG

The direction flag (DF, located in bit 10 of the EFLAGS register) controls the string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (that is, to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

3.4.4. System Flags and IOPL Field

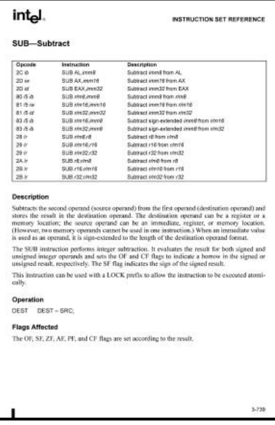
The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. **They should not be modified by application programs.** The functions of the system flags are as follows:

3-14

Summary of ADD Instruction

- Basic Function:
  - Adds source operand to destination operand.
  - Both signed and unsigned addition performed.
- Addressing Modes:
  - Source operand can be immediate, a register or memory.
  - Destination operand can be a register or memory.
  - Source and destination cannot both be memory.
- Flags Affected:
  - OF = 1 if two's complement overflow occurred
  - SF = 1 if result in two's complement is negative (MSbit = 1)
  - ZF = 1 if result is zero
  - CF = 1 if unsigned overflow occurred

19 UMBC, CMSC313, Richard Chang <chang@umbc.edu>



INSTRUCTION SET REFERENCE

SUB—Subtract

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

Description

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

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

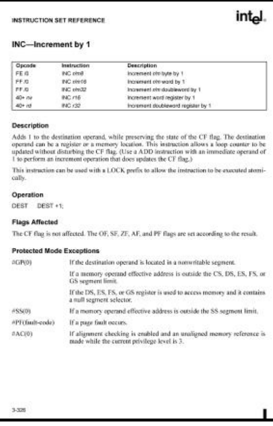
```
DEST    DEST – SRC;
```

Flags Affected

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

3-739

20



INSTRUCTION SET REFERENCE

INC—Increment by 1

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

Description

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

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST    DEST +1;
```

Flags Affected

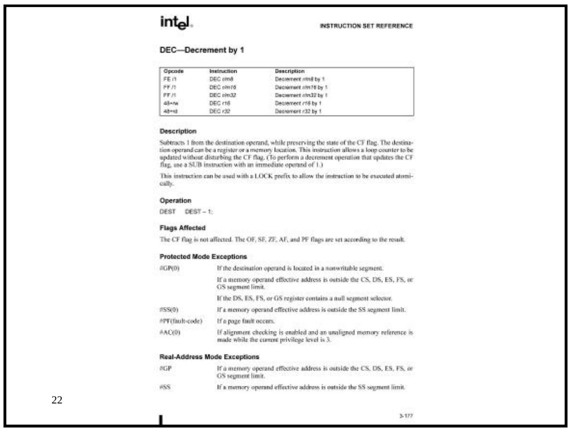
The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

3-326

21



INSTRUCTION SET REFERENCE

DEC—Decrement by 1

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

Description

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

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST    DEST – 1;
```

Flags Affected

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

Protected Mode Exceptions

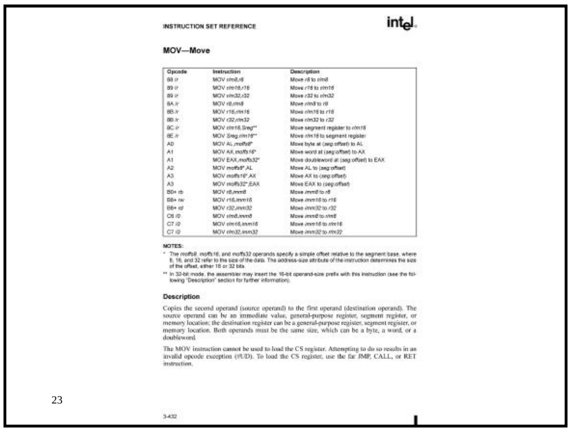
#GP(0)	If the destination operand is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

3-177

22



INSTRUCTION SET REFERENCE

MOV—Move

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

NOTES:

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

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

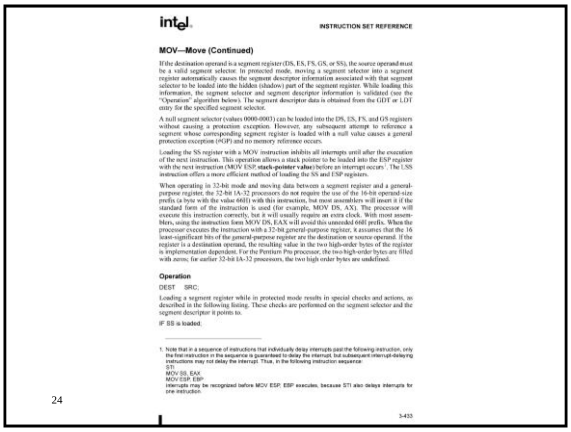
Description

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

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

3-432

23



INSTRUCTION SET REFERENCE

MOV—Move (Continued)

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

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

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

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

Operation

```
DEST    SRC;
```

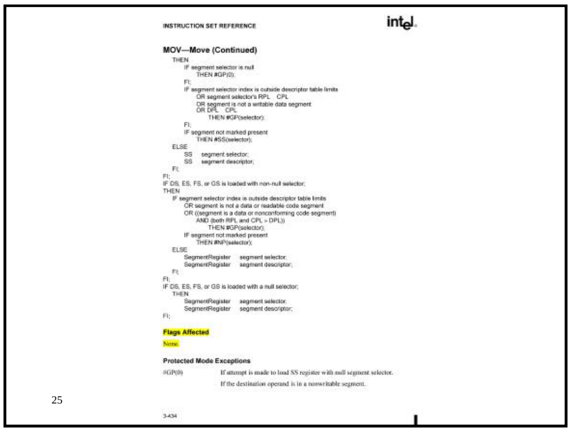
Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
```

1. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:
   STI
   MOV SS, EAX
   MOV ESP, EBP
   interrupts may be recognized before MOV ESP, EBP executes, because STI also delays interrupts for one instruction.

3-433

24


INSTRUCTION SET REFERENCE

MOV—Move (Continued)

```
    THEN
        IF segment selector is null
            THEN #GP(0);
        FI;
        IF segment selector index is outside descriptor table limits
            OR segment selector's RPL    CPL
            OR segment is not a writable data segment
            OR DPL    CPL
                THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #SS(selector);
    ELSE
        SS    segment selector;
        SS    segment descriptor;
    FI;
FI;
IF DS, ES, FS, or GS is loaded with non-null selector;
THEN
    IF segment selector index is outside descriptor table limits
        OR segment is not a data or readable code segment
        OR ((segment is a data or nonconforming code segment)
            AND (both RPL and CPL > DPL))
                THEN #GP(selector);
        IF segment not marked present
            THEN #NP(selector);
    ELSE
        SegmentRegister    segment selector;
        SegmentRegister    segment descriptor;
    FI;
FI;
IF DS, ES, FS, or GS is loaded with a null selector;
    THEN
        SegmentRegister    segment selector;
        SegmentRegister    segment descriptor;
FI;
```

Flags Affected

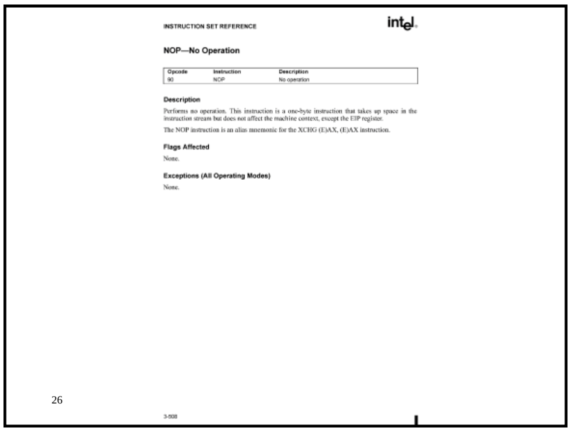
None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination operand is in a nonwritable segment.

3-434

25


INSTRUCTION SET REFERENCE

NOP—No Operation

Opcode	Instruction	Description
90	NOP	No operation

Description

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

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

Flags Affected

None.

Exceptions (All Operating Modes)

None.

3-508

26

Conditional Jumps

27

Branching Instructions

- JMP = unconditional jump
- Conditional jumps use the flags to decide whether to jump to the given label or to continue.
- The flags were modified by previous arithmetic instructions or by a compare (CMP) instruction.
- The instruction:

  CMP op1, op2

  computes the unsigned and two's complement subtraction op1 - op2 and modifies the flags. The contents of op1 are not affected.

28 UMBC, CMSC313, Richard Chang <chang@umbc.edu>

Example of CMP instruction

- Suppose AL contains 254. After the instruction:

  CMP AL, 17

  CF = 0, OF = 0, SF = 1 and ZF = 0.
- A JA (jump above) instruction would jump.
- A JG (jump greater than) instruction wouldn't jump.
- Both signed and unsigned comparisons use the same CMP instruction.
- Signed and unsigned jump instructions interpret the flags differently.

29 UMBC, CMSC313, Richard Chang <chang@umbc.edu>

More Conditional Jumps

- Uses flags to determine whether to jump
  - Example: JAE (jump above or equal) jumps when the Carry Flag = 0

```
CMP EAX, 1492
JAE OceanBlue
```

- Unsigned vs signed jumps
  - Example: use JAE for unsigned data JGE (greater than or equal) for signed data

```
CMP EAX, 1492        CMP EAX, -42
JAE OceanBlue        JGE Somewhere
```

30 UMBC, CMSC313, Richard Chang <chang@umbc.edu>

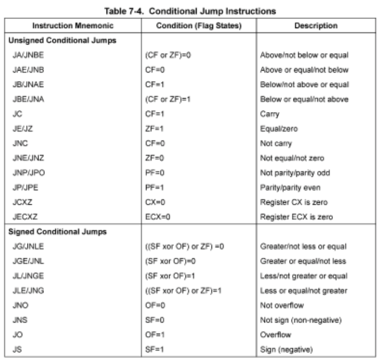
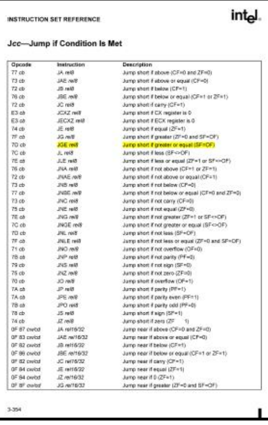


Table 7-4. Conditional Jump Instructions

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

31



INSTRUCTION SET REFERENCE

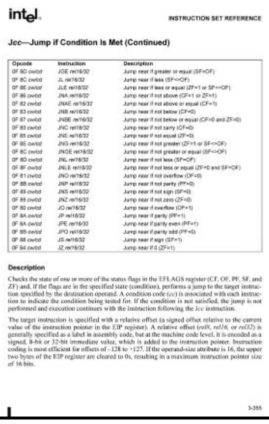
intel.

Jcc—Jump if Condition Is Met

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

32

3-354



intel.

INSTRUCTION SET REFERENCE

Jcc—Jump if Condition Is Met (Continued)

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

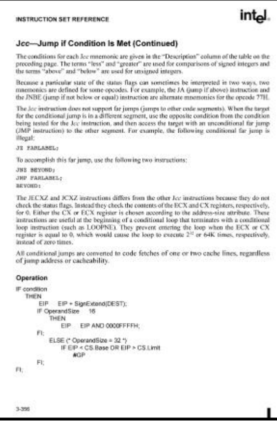
Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

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

33

3-355



INSTRUCTION SET REFERENCE

Jcc—Jump if Condition Is Met (Continued)

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

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

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

```
JZ FARLABEL;
```

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

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

The JECXZ and JCXZ instructions differs from the other *Jcc* instructions because they do not check the status flags. Instead they check the contents of the ECX and CX registers, respectively, for 0. Either the CX or ECX register is chosen according to the address-size attribute. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE). They prevent entering the loop when the ECX or CX register is equal to 0, which would cause the loop to execute 2^{32} or 64K times, respectively, instead of zero times.

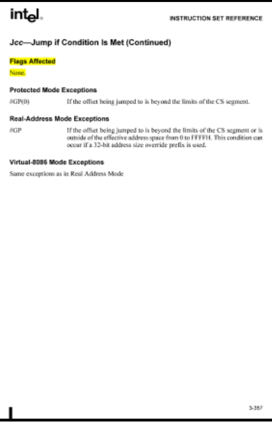
All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF condition
    THEN
        EIP    EIP + SignExtend(DEST);
        IF OperandSize    16
            THEN
                EIP    EIP AND 0000FFFFH;
        FI;
            ELSE (* OperandSize = 32 *)
                IF EIP < CS.Base OR EIP > CS.Limit
                    #GP
        FI;
FI;
```

3-356

34



INSTRUCTION SET REFERENCE

Jcc—Jump if Condition Is Met (Continued)

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

3-357

35

Closer look at JGE

- JGE jumps if and only if SF = OF
  - Examples using 8-bit registers. Which of these result in a jump?

```
1. MOV AL, 96          2. MOV AL, -64
   CMP AL, 80             CMP AL, 80
   JGE Somewhere          JGE Somewhere

3. MOV AL, 64          4. MOV AL, 64
   CMP AL, -80            CMP AL, 80
   JGE Somewhere          JGE Somewhere
```

- If OF=0, then use SF to check whether A-B >= 0.
- If OF=1, then do opposite of SF.
- JGE works after a CMP instruction, even when subtracting the operands result in an overflow!

36

Short Jumps vs. Near Jumps

- Jumps use relative addressing
  - assembler computes an *offset* from address of current instruction.
  - produces *relocatable* code
- SHORT jumps use 8-bit offsets
  - target label within -128 bytes to +127 bytes
- NEAR jumps use 32-bit offsets
  - target label within -2^{31} bytes to $+2^{31}-1$ bytes

  (there is also an absolute address version)

37

Short Jumps vs. Near Jumps

- Some assemblers determine SHORT vs NEAR jumps automatically, but *some do not*.
- explicitly specify SHORT jumps

```
jmp  SHORT somewhere
```

- explicitly specify NEAR jumps

```
jge  NEAR somewhere
```

38

```
; File: jmp.asm
;
; Demonstrating near and short jumps
;

        section .text
        global _start

_start: nop

        ; initialize

start:  mov     eax, 17         ; eax := 17
        cmp     eax, 42         ; 17 - 42 is ...

        jge     exit            ; exit if 17 >= 42
        jge     short exit
        jge     near exit

        jmp     exit
        jmp     short exit
        jmp     near exit

exit:   mov     ebx, 0          ; exit code, 0=normal
        mov     eax, 1          ; Exit.
        int     080H            ; Call kernel.
```

```
 1                          ; File: jmp.asm
 2                          ;
 3                          ; Demonstrating near and short jumps
 4                          ;
 5
 6                                  section .text
 7                                  global _start
 8
 9 00000000 90              _start: nop
10
11                                  ; initialize
12
13 00000001 B811000000      start:  mov     eax, 17         ; eax := 17
14 00000006 3D2A000000              cmp     eax, 42         ; 17 - 42 is ...
15
16 0000000B 7D14                    jge     exit            ; exit if 17 >= 42
17 0000000D 7D12                    jge     short exit
18 0000000F 0F8D0C000000            jge     near exit
19
20 00000015 E907000000              jmp     exit
21 0000001A EB05                    jmp     short exit
22 0000001C E900000000              jmp     near exit
23
24 00000021 BB00000000      exit:   mov     ebx, 0          ; exit code, 0=normal
25 00000026 B801000000              mov     eax, 1          ; Exit.
26 0000002B CD80                    int     080H            ; Call kernel.
```

Using Jump Instructions

Converting an if Statement

```
if (x < y) {
    statement block 1 ;
} else {
    statement block 2 ;
}
```

```
    MOV EAX,[x]
    CMP EAX,[y]
    JGE ElsePart
     .                ; if part
     .                ; statement block 1
     .
    JMP Done          ; skip over else part
ElsePart:
     .                ; else part
     .                ; statement block 2
     .
Done:
```

Converting a while Loop

```
while (i > 0) {
    statement 1 ;
    statement 2 ;
}
```

```
WhileTop:
    MOV EAX,[i]
    CMP EAX,0
    JLE Done
     .                ; statement 1
     .
     .
     .                ; statement 2
     .
     .
    JMP WhileTop
Done:
```

43

References

- **Some figures and diagrams from *IA-32 Intel Architecture Software Developer's Manual, Vols 1-3***

  <http://developer.intel.com/design/Pentium4/manuals/>

44