

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP1 EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 ² [BX]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[BX+SI]+disp8 ³ [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =	AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111					
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)										
[EAX] [ECX] [EDX] [EBX] [--][--] ¹ disp32 ² [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F			
[EAX]+disp8 ³ [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8		01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F		
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32			10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF	
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7				11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4 and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

NOTES:

- The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits Effective Address

- | | |
|----|---------------------------------|
| 00 | [scaled index] + disp32 |
| 01 | [scaled index] + disp8 + [EBP] |
| 10 | [scaled index] + disp32 + [EBP] |

2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- 64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH ES ⁶⁴	POP ES ⁶⁴
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH SS ⁶⁴	POP SS ⁶⁴
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=ES (Prefix)	DAA ⁶⁴
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=SS (Prefix)	AAA ⁶⁴
4	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA ⁶⁴ / PUSHAD ⁶⁴	POPA ⁶⁴ / POPAD ⁶⁴	BOUND ⁶⁴ Gv, Ma	ARPL ⁶⁴ Ew, Gw MOVSD ⁶⁴ Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Eb, lb	Ev, lz	Eb, lb ⁶⁴	Ev, lb	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	AL/R8L, lb	CL/R9L, lb	DL/R10L, lb	BL/R11L, lb	AH/R12L, lb	CH/R13L, lb	DH/R14L, lb	BH/R15L, lb
C	Eb, lb	Ev, lb	near RET ⁶⁴ lw	near RET ⁶⁴	LES ⁶⁴ Gz, Mp VEX+2byte	LDS ⁶⁴ Gz, Mp VEX+1byte	Grp 11 ^{1A} - MOV Eb, lb	Ev, lz
D	Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM ⁶⁴ lb	AAD ⁶⁴ lb		XLAT/ XLATB
E	LOOPNE ⁶⁴ / LOOPNZ ⁶⁴ Jb	LOOPE ⁶⁴ / LOOPZ ⁶⁴ Jb	LOOP ⁶⁴ Jb	JRCXZ ⁶⁴ Jb	IN AL, lb	eAX, lb	OUT lb, AL	lb, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3 ^{1A} Eb	Ev

Table A-2. One-byte Opcode Map: (08H — FFH) *

	8	9	A	B	C	D	E	F
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH CS ⁱ⁶⁴	2-byte escape (Table A-3)
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH DS ⁱ⁶⁴	POP DS ⁱ⁶⁴
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=CS (Prefix)	DAS ⁱ⁶⁴
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=DS (Prefix)	AAS ⁱ⁶⁴
4	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB
5	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH ^{d64} lz	IMUL Gv, Ev, lz	PUSH ^{d64} lb	IMUL Gv, Ev, lb	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A ^{1A} POP ^{d64} Ev
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL ⁱ⁶⁴ Ap	FWAIT/ WAIT	PUSHF/D/Q ^{d64} / Fv	POPF/D/Q ^{d64} / Fv	SAHF	LAHF
A	TEST AL, lb	rAX, lz	STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Yv
B	rAX/r8, lv	rCX/r9, lv	rDX/r10, lv	rBX/r11, lv	rSP/r12, lv	rBP/r13, lv	rSI/r14, lv	rDI/r15, lv
C	ENTER lw, lb	LEAVE ^{d64}	far RET lw	far RET	INT 3	INT lb	INTO ⁱ⁶⁴	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	near CALL ^{f64} Jz	near ^{f64} Jz	JMP far ^{f64} Ap	short ^{f64} Jb	AL, DX	eAX, DX	DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 ^{1A}	INC/DEC Grp 5 ^{1A}

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) *

	pxf	0	1	2	3	4	5	6	7
0		Grp 6 ^{1A}	Grp 7 ^{1A}	LAR Gv, Ew	LSL Gv, Ew		SYSCALL ⁰⁶⁴	CLTS	SYSRET ⁰⁶⁴
1		vmovups Vps, Wps	vmovups Wps, Vps	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps ^{v1} Vdq, Hq, Mq vmovhlps Vdq, Hq, Uq	vmovhps ^{v1} Mq, Vq
	66	vmovupd Vpd, Wpd	vmovupd Wpd, Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx, Hx, Wx	vunpckhpd Vx, Hx, Wx	vmovhpd ^{v1} Vdq, Hq, Mq	vmovhpd ^{v1} Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2		MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3		WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4		CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy, Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12 ^{1A})	(Grp 13 ^{1A})	(Grp 14 ^{1A})	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper ^v vzeroall ^v
	66	vpshufd Vx, Wx, Ib				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshufhw Vx, Wx, Ib							
	F2	vpshufw Vx, Wx, Ib							

Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

	pxf	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2 ^{1B}		prefetchw(/1) Ev		
1		Prefetch ^{1C} (Grp 16 ^{1A})							NOP /0 Ev
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd, Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtss2ss Vss, Hss, Ey		vcvtss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtss2sd Vsd, Hsd, Ey		vcvtss2si Gy, Wsd	vcvtss2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvtss2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtss2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpcklqdq Vx, Hx, Wx	vpunpckhqdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsubpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx, Vx
	F3							vmovq Vq, Wq	vmovdqu Wx, Vx
	F2					vhaddps Vps, Hps, Wps	vhsubps Vps, Hps, Wps		

Table A-3. Two-byte Opcode Map: 80H — F7H (First Byte is 0FH) *

	pxf	0	1	2	3	4	5	6	7
8		Jcc ⁶⁴ , Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH ^{d64} FS	POP ^{d64} FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG Eb, Gb		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb Gv, Ew	
C		XADD Eb, Gb	XADD Ev, Gv	vcmpps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9 ^{1A}
	66			vcmpsd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib	
	F3			vcmpss Vss,Hss,Wss,Ib					
	F2			vcmpsdb Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vx, Hx, Wx	vpsraw Vx, Hx, Wx	vpsrad Vx, Hx, Wx	vpavgw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvtq2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtq2pd Vx, Wpd	
	F2							vcvtq2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) *

	pxf	8	9	A	B	C	D	E	F
8		Jcc ⁶⁴ , Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH ^{d64} GS	POP ^{d64} GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 ^{1A}) ^{1C}	IMUL Gv, Ev
B		JMPE (reserved for emulator on IPF)	Grp 10 ^{1A} Invalid Opcode ^{1B}	Grp 8 ^{1A} Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSBX Gv, Eb	Gv, Ew
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
D		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmaxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpminsw Vx, Hx, Wx	vpor Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmaxsw Vx, Hx, Wx	vpxor Vx, Hx, Wx
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) *

	pxf	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phadd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vx, Hx, Wx	vphaddw Vx, Hx, Wx	vphadd Vx, Hx, Wx	vphaddsw Vx, Hx, Wx	vpaddubsw Vx, Hx, Wx	vphsubw Vx, Hx, Wx	vphsubd Vx, Hx, Wx	vphsubsw Vx, Hx, Wx
1	66	pblendvb Vdq, Wdq			vcvtph2ps ^v Vx, Wx, Ib	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps ^v Vqq, Hqq, Wqq	vptest Vx, Wx
	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxwd Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
2	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxwd Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd ^v Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
3	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q ^v Vx, Hx, Wx	vpsravd ^v Vx, Hx, Wx	vpslvd/q ^v Vx, Hx, Wx
4									
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
	66	vgatherdd/q ^v Vx,Hx,Wx	vgatherqd/q ^v Vx,Hx,Wx	vgatherdps/d ^v Vx,Hx,Wx	vgatherqps/d ^v Vx,Hx,Wx			vfmaddsub132ps/d ^v Vx,Hx,Wx	vfmsubadd132ps/d ^v Vx,Hx,Wx
A	66							vfmaddsub213ps/d ^v Vx,Hx,Wx	vfmsubadd213ps/d ^v Vx,Hx,Wx
B	66							vfmaddsub231ps/d ^v Vx,Hx,Wx	vfmsubadd231ps/d ^v Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN ^v Gy, By, Ey	Grp 17 ^{1A}		BZHI ^v Gy, Ey, By		BEXTR ^v Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX ^v Gy, Ey, By
	F3						PEXT ^v Gy, By, Ey	ADOX Gy, Ey	SARX ^v Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP ^v Gy, By, Ey	MULX ^v By,Gy,rDX,Ey	SHRX ^v Gy, Ey, By
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

Table A-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H) *

	pxf	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vx, Hx, Wx	vpsignw Vx, Hx, Wx	vpsignd Vx, Hx, Wx	vpmulhrsw Vx, Hx, Wx	vpermilps ^V Vx,Hx,Wx	vpermilpd ^V Vx,Hx,Wx	vtestps ^V Vx, Wx	vtestpd ^V Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss ^V Vx, Wd	vbroadcastsd ^V Vqq, Wq	vbroadcastf128 ^V Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2	66	vpmuldq Vx, Hx, Wx	vpcmpqq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps ^V Vx,Hx,Mx	vmaskmovpd ^V Vx,Hx,Mx	vmaskmovps ^V Mx,Hx,Vx	vmaskmovpd ^V Mx,Hx,Vx
3	66	vpminsb Vx, Hx, Wx	vpminsd Vx, Hx, Wx	vpminuw Vx, Hx, Wx	vpminud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
4									
5	66	vpbroadcastd ^V Vx, Wx	vpbroadcastq ^V Vx, Wx	vpbroadcasti128 ^V Vqq, Mdq					
6									
7	66	vpbroadcastb ^V Vx, Wx	vpbroadcastw ^V Vx, Wx						
8	66					vpmaskmovd/q ^V Vx,Hx,Mx		vpmaskmovd/q ^V Mx,Vx,Hx	
9	66	vfmadd132ps/d ^V Vx, Hx, Wx	vfmadd132ss/d ^V Vx, Hx, Wx	vfmsub132ps/d ^V Vx, Hx, Wx	vfmsub132ss/d ^V Vx, Hx, Wx	vfnmadd132ps/d ^V Vx, Hx, Wx	vfnmadd132ss/d ^V Vx, Hx, Wx	vfnmsub132ps/d ^V Vx, Hx, Wx	vfnmsub132ss/d ^V Vx, Hx, Wx
A	66	vfmadd213ps/d ^V Vx, Hx, Wx	vfmadd213ss/d ^V Vx, Hx, Wx	vfmsub213ps/d ^V Vx, Hx, Wx	vfmsub213ss/d ^V Vx, Hx, Wx	vfnmadd213ps/d ^V Vx, Hx, Wx	vfnmadd213ss/d ^V Vx, Hx, Wx	vfnmsub213ps/d ^V Vx, Hx, Wx	vfnmsub213ss/d ^V Vx, Hx, Wx
B	66	vfmadd231ps/d ^V Vx, Hx, Wx	vfmadd231ss/d ^V Vx, Hx, Wx	vfmsub231ps/d ^V Vx, Hx, Wx	vfmsub231ss/d ^V Vx, Hx, Wx	vfnmadd231ps/d ^V Vx, Hx, Wx	vfnmadd231ss/d ^V Vx, Hx, Wx	vfnmsub231ps/d ^V Vx, Hx, Wx	vfnmsub231ss/d ^V Vx, Hx, Wx
C									
D	66				VAESIMC Vdq, Wdq	VAESEC Vdq,Hdq,Wdq	VAESENCLAST Vdq,Hdq,Wdq	VAESDEC Vdq,Hdq,Wdq	VAESDECLAST Vdq,Hdq,Wdq
E									
F									
	66								
	F3								
	F2								
	66 & F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *

	px	0	1	2	3	4	5	6	7
0	66	vpermq ^v Vqq, Wqq, lb	vpermpd ^v Vqq, Wqq, lb	vpblendd ^v Vx, Hx, Wx, lb		vpermilps ^v Vx, Wx, lb	vpermilpd ^v Vx, Wx, lb	vperm2f128 ^v Vqq, Hqq, Wqq, lb	
1	66					vpextrb Rd/Mb, Vdq, lb	vpextrw Rd/Mw, Vdq, lb	vpextrd/q Ey, Vdq, lb	vextractps Ed, Vdq, lb
2	66	vpinsrb Vdq, Hdq, Ry/Mb, lb	vinsertps Vdq, Hdq, Udq/Md, lb	vpinsrd/q Vdq, Hdq, Ey, lb					
3									
4	66	vdpps Vx, Hx, Wx, lb	vdppd Vdq, Hdq, Wdq, lb	vmepsadbw Vx, Hx, Wx, lb		vpcimulqdq Vdq, Hdq, Wdq, lb		vperm2i128 ^v Vqq, Hqq, Wqq, lb	
5									
6	66	vpcmpestrm Vdq, Wdq, lb	vpcmpestri Vdq, Wdq, lb	vpcmpistrm Vdq, Wdq, lb	vpcmpistri Vdq, Wdq, lb				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX ^v Gy, Ey, lb							

Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) *

	px	8	9	A	B	C	D	E	F
0									palgnr Pq, Qq, Ib
	66	vroundps Vx, Wx, Ib	vroundpd Vx, Wx, Ib	vroundss Vss, Wss, Ib	vroundsd Vsd, Wsd, Ib	vblendps Vx, Hx, Wx, Ib	vblendpd Vx, Hx, Wx, Ib	vpblendw Vx, Hx, Wx, Ib	vpalignr Vx, Hx, Wx, Ib
1	66	vinserff128 ^v Vqq, Hqq, Wqq, Ib	vextractf128 ^v Wdq, Vqq, Ib				vcvtps2ph ^v Wx, Vx, Ib		
2									
3	66	vinserff128 ^v Vqq, Hqq, Wqq, Ib	vextractf128 ^v Wdq, Vqq, Ib						
4	66			vblendvps ^v Vx, Hx, Wx, Lx	vblendvpd ^v Vx, Hx, Wx, Lx	vpblendvb ^v Vx, Hx, Wx, Lx			
5									
6									
7									
8									
9									
A									
B									
C									
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B		POP							
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B		TEST lb/lz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL ¹⁶⁴ Ev	far CALL Ep	near JMP ¹⁶⁴ Ev	far JMP Mp	PUSH ¹⁶⁴ Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110)					SWAPGS ¹⁶⁴ (000) RDTSCP (001)
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	VMPTRST Mq
		11B								RDRAND Rv	RDSEED Rv
0F B9	10	mem									
		11B									
C6	11	mem		MOV Eb, lb							
		11B									XABORT (000) lb
C7		mem		MOV Ev, lz							
		11B									XBEGIN (000) Jz
0F 71	12	mem									
		11B				psrlw Nq, lb		psraw Nq, lb		psllw Nq, lb	
		66				vpsrlw Hx,Ux,lb		vpsraw Hx,Ux,lb		vpsllw Hx,Ux,lb	
0F 72	13	mem									
		11B				psrld Nq, lb		psrad Nq, lb		pslld Nq, lb	
		66				vpsrld Hx,Ux,lb		vpsrad Hx,Ux,lb		vpslld Hx,Ux,lb	
0F 73	14	mem									
		11B				psrlq Nq, lb				psllq Nq, lb	
		66				vpsrlq Hx,Ux,lb	vpsrldq Hx,Ux,lb			vpsllq Hx,Ux,lb	vpslldq Hx,Ux,lb

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group * (Contd.)

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
									lfence	mfence	sfence
		11B	F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry				
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B									
VEX.0F38 F3	17	mem			BLSR ^v By, Ey	BLSMSK ^v By, Ey	BLSI ^v By, Ey				
		11B									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

NOTES:

- * All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	EXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMPP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
37	AAA	NP	Invalid	Valid	ASCII adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```

IF 64-Bit Mode
  THEN
    #UD;
  ELSE
    IF ((AL AND 0FH) > 9) or (AF = 1)
      THEN
        AX ← AX + 106H;
        AF ← 1;
        CF ← 1;
      ELSE
        AF ← 0;
        CF ← 0;
    FI;
    AL ← AL AND 0FH;
  FI;

```

Flags Affected

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

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
D5 0A	AAD	NP	Invalid	Valid	ASCII adjust AX before division.
D5 <i>ib</i>	AAD <i>imm8</i>	NP	Invalid	Valid	Adjust AX before division to number base <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

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

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
  THEN
    #UD;
  ELSE
    tempAL ← AL;
    tempAH ← AH;
    AL ← (tempAL + (tempAH * imm8)) AND FFH;
    (* imm8 is set to 0AH for the AAD mnemonic.*)
    AH ← 0;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
D4 0A	AAM	NP	Invalid	Valid	ASCII adjust AX after multiply.
D4 <i>ib</i>	AAM <i>imm8</i>	NP	Invalid	Valid	Adjust AX after multiply to number base <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL ← AL;
        AH ← tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
        AL ← tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#DE If an immediate value of 0 is used.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
3F	AAS	NP	Invalid	Valid	ASCII adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```

IF 64-bit mode
  THEN
    #UD;
  ELSE
    IF ((AL AND 0FH) > 9) or (AF = 1)
      THEN
        AX ← AX - 6;
        AH ← AH - 1;
        AF ← 1;
        CF ← 1;
        AL ← AL AND 0FH;
      ELSE
        CF ← 0;
        AF ← 0;
        AL ← AL AND 0FH;
    FI;
  FI;

```

Flags Affected

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

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

ADC—Add with Carry

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
14 <i>ib</i>	ADC AL, <i>imm8</i>	I	Valid	Valid	Add with carry <i>imm8</i> to AL.
15 <i>iw</i>	ADC AX, <i>imm16</i>	I	Valid	Valid	Add with carry <i>imm16</i> to AX.
15 <i>id</i>	ADC EAX, <i>imm32</i>	I	Valid	Valid	Add with carry <i>imm32</i> to EAX.
REX.W + 15 <i>id</i>	ADC RAX, <i>imm32</i>	I	Valid	N.E.	Add with carry <i>imm32</i> sign extended to 64-bits to RAX.
80 /2 <i>ib</i>	ADC <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Add with carry <i>imm8</i> to <i>r/m8</i> .
REX + 80 /2 <i>ib</i>	ADC <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Add with carry <i>imm8</i> to <i>r/m8</i> .
81 /2 <i>iw</i>	ADC <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Add with carry <i>imm16</i> to <i>r/m16</i> .
81 /2 <i>id</i>	ADC <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Add with CF <i>imm32</i> to <i>r/m32</i> .
REX.W + 81 /2 <i>id</i>	ADC <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Add with CF <i>imm32</i> sign extended to 64-bits to <i>r/m64</i> .
83 /2 <i>ib</i>	ADC <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> to <i>r/m16</i> .
83 /2 <i>ib</i>	ADC <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> into <i>r/m32</i> .
REX.W + 83 /2 <i>ib</i>	ADC <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Add with CF sign-extended <i>imm8</i> into <i>r/m64</i> .
10 /r	ADC <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Add with carry byte register to <i>r/m8</i> .
REX + 10 /r	ADC <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Add with carry byte register to <i>r/m64</i> .
11 /r	ADC <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Add with carry <i>r16</i> to <i>r/m16</i> .
11 /r	ADC <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Add with CF <i>r32</i> to <i>r/m32</i> .
REX.W + 11 /r	ADC <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Add with CF <i>r64</i> to <i>r/m64</i> .
12 /r	ADC <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Add with carry <i>r/m8</i> to byte register.
REX + 12 /r	ADC <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Add with carry <i>r/m64</i> to byte register.
13 /r	ADC <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Add with carry <i>r/m16</i> to <i>r16</i> .
13 /r	ADC <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Add with CF <i>r/m32</i> to <i>r32</i> .
REX.W + 13 /r	ADC <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Add with CF <i>r/m64</i> to <i>r64</i> .

NOTES:

*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX	<i>imm8</i>	NA	NA

Description

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

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

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

$DEST \leftarrow DEST + SRC + CF;$

Intel C/C++ Compiler Intrinsic Equivalent

ADC: extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);

ADC: extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);

ADC: extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);

ADC: extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
REX.w + 66 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

IF OperandSize is 64-bit

THEN CF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + CF;

ELSE CF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + CF;

FI;

Flags Affected

CF is updated based on result. OF, SF, ZF, AF and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

```
unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
```

```
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);
```

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD If the LOCK prefix is used.

	If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	I	Valid	Valid	Add <i>imm8</i> to AL.
05 <i>iw</i>	ADD AX, <i>imm16</i>	I	Valid	Valid	Add <i>imm16</i> to AX.
05 <i>id</i>	ADD EAX, <i>imm32</i>	I	Valid	Valid	Add <i>imm32</i> to EAX.
REX.W + 05 <i>id</i>	ADD RAX, <i>imm32</i>	I	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to RAX.
80 /0 <i>ib</i>	ADD <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to <i>r/m8</i> .
REX + 80 /0 <i>ib</i>	ADD <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to <i>r/m64</i> .
81 /0 <i>iw</i>	ADD <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to <i>r/m16</i> .
81 /0 <i>id</i>	ADD <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to <i>r/m32</i> .
REX.W + 81 /0 <i>id</i>	ADD <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to <i>r/m64</i> .
83 /0 <i>ib</i>	ADD <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to <i>r/m16</i> .
83 /0 <i>ib</i>	ADD <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to <i>r/m32</i> .
REX.W + 83 /0 <i>ib</i>	ADD <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to <i>r/m64</i> .
00 /r	ADD <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Add <i>r8</i> to <i>r/m8</i> .
REX + 00 /r	ADD <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Add <i>r8</i> to <i>r/m8</i> .
01 /r	ADD <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Add <i>r16</i> to <i>r/m16</i> .
01 /r	ADD <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Add <i>r32</i> to <i>r/m32</i> .
REX.W + 01 /r	ADD <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Add <i>r64</i> to <i>r/m64</i> .
02 /r	ADD <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Add <i>r/m8</i> to <i>r8</i> .
REX + 02 /r	ADD <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Add <i>r/m8</i> to <i>r8</i> .
03 /r	ADD <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Add <i>r/m16</i> to <i>r16</i> .
03 /r	ADD <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Add <i>r/m32</i> to <i>r32</i> .
REX.W + 03 /r	ADD <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Add <i>r/m64</i> to <i>r64</i> .

NOTES:

*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>r</i> , <i>w</i>)	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA
MR	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	ModRM:reg (<i>r</i>)	NA	NA
MI	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX	<i>imm8</i>	NA	NA

Description

Adds the destination operand (first operand) and the source operand (second operand) and then 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.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

$DEST \leftarrow DEST + SRC;$

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 58 /r ADDPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 58 /r VADDPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 58 /r VADDPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Add packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of SIMD double-precision floating-point operation.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ADDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
DEST[127:64] ← DEST[127:64] + SRC[127:64];
DEST[VLMAX-1:128] (Unmodified)
```

VADDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```


VADDPD (VEX.256 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] + SRC2[63:0]$

$DEST[127:64] \leftarrow SRC1[127:64] + SRC2[127:64]$

$DEST[191:128] \leftarrow SRC1[191:128] + SRC2[191:128]$

$DEST[255:192] \leftarrow SRC1[255:192] + SRC2[255:192]$

Intel C/C++ Compiler Intrinsic Equivalent

ADDPD: `__m128d _mm_add_pd (__m128d a, __m128d b)`

VADDPD: `__m256d _mm256_add_pd (__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5B /r ADDPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> and stores result in <i>xmm1</i> .
VEX.NDS.128.OF.WIG 5B /r VADDPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add packed single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.OF.WIG 5B /r VADDPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Add packed single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of SIMD single-precision floating-point operation.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ADDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[63:32] ← DEST[63:32] + SRC[63:32];
DEST[95:64] ← DEST[95:64] + SRC[95:64];
DEST[127:96] ← DEST[127:96] + SRC[127:96];
DEST[VLMAX-1:128] (Unmodified)
```

VADDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VADDPS (VEX.256 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0]$
 $DEST[63:32] \leftarrow SRC1[63:32] + SRC2[63:32]$
 $DEST[95:64] \leftarrow SRC1[95:64] + SRC2[95:64]$
 $DEST[127:96] \leftarrow SRC1[127:96] + SRC2[127:96]$
 $DEST[159:128] \leftarrow SRC1[159:128] + SRC2[159:128]$
 $DEST[191:160] \leftarrow SRC1[191:160] + SRC2[191:160]$
 $DEST[223:192] \leftarrow SRC1[223:192] + SRC2[223:192]$
 $DEST[255:224] \leftarrow SRC1[255:224] + SRC2[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

ADDPS: `__m128 _mm_add_ps(__m128 a, __m128 b)`

VADDPS: `__m256 _mm256_add_ps (__m256 a, __m256 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDSD—Add Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5B /r ADDSD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE2	Add the low double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5B /r VADDSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m64</i>	RVM	V/V	AVX	Add the low double-precision floating-point value from <i>xmm3/mem</i> to <i>xmm2</i> and store the result in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the double-precision floating-point result in the destination operand.

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

ADDSD (128-bit Legacy SSE version)

$DEST[63:0] \leftarrow DEST[63:0] + SRC[63:0]$

$DEST[VLMAX-1:64]$ (Unmodified)

VADDSD (VEX.128 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] + SRC2[63:0]$

$DEST[127:64] \leftarrow SRC1[127:64]$

$DEST[VLMAX-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSD: `__m128d _mm_add_sd (m128d a, m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

ADDSS—Add Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Add the low single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 58 /r VADDSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Add the low single-precision floating-point value from <i>xmm3/mem</i> to <i>xmm2</i> and store the result in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the low single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the single-precision floating-point result in the destination operand.

The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

ADDSS DEST, SRC (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow DEST[31:0] + SRC[31:0];$

$DEST[VLMAX-1:32]$ (Unmodified)

VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0]$

$DEST[127:32] \leftarrow SRC1[127:32]$

$DEST[VLMAX-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSS: `__m128 _mm_add_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

ADDSUBPD—Packed Double-FP Add/Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F D0 /r ADDSUBPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE3	Add/subtract double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG D0 /r VADDSUBPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add/subtract packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG D0 /r VADDSUBPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Add / subtract packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double-precision floating-point values from the second source operand from the corresponding double-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

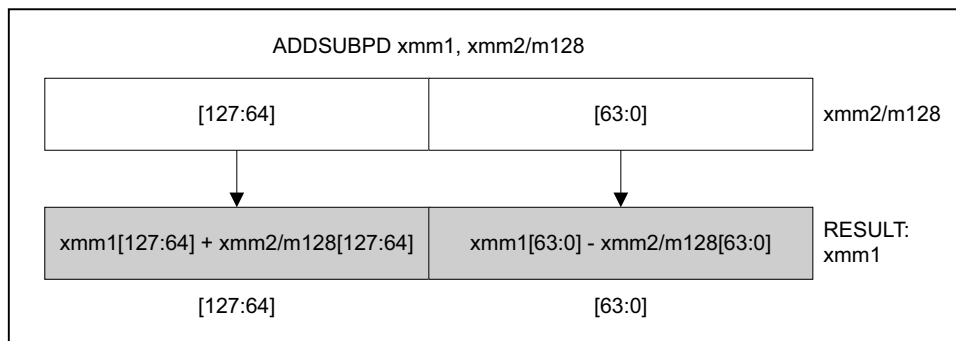


Figure 3-3. ADDSUBPD—Packed Double-FP Add/Subtract

Operation

ADDSUBPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VADDSUBPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VADDSUBPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] - \text{SRC2}[191:128]$
 $\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] + \text{SRC2}[255:192]$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD: `__m128d _mm_addsub_pd(__m128d a, __m128d b)`

VADDSUBPD: `__m256d _mm256_addsub_pd (__m256d a, __m256d b)`

Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDSUBPS—Packed Single-FP Add/Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F D0 /r ADDSUBPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE3	Add/subtract single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG D0 /r VADDSUBPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add/subtract single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.F2.0F.WIG D0 /r VADDSUBPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Add / subtract single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

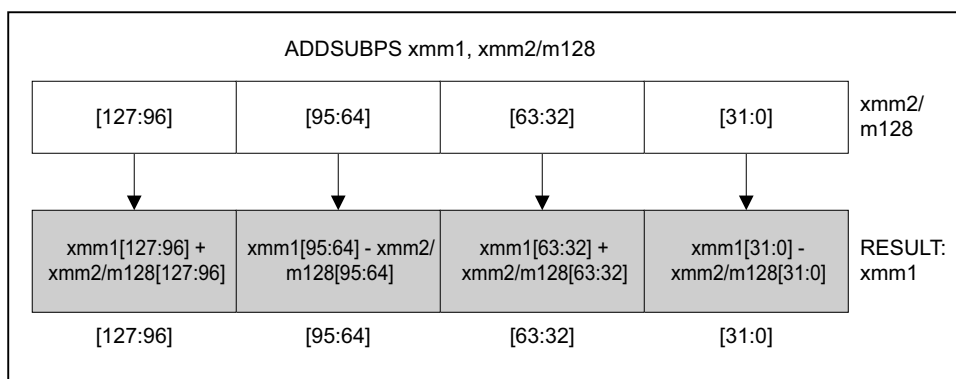
Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.



OM15992

Figure 3-4. ADDSUBPS—Packed Single-FP Add/Subtract

Operation

ADDSUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] - \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{DEST}[63:32] + \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{DEST}[95:64] - \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{DEST}[127:96] + \text{SRC}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VADDSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VADDSUBPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] - \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS: `__m128_mm_addsub_ps(__m128 a, __m128 b)`

VADDSUBPS: `__m256_mm256_addsub_ps (__m256 a, __m256 b)`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F6 /r ADOX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with OF, r/m32 to r32, writes OF.
REX.w + F3 0F 38 F6 /r ADOX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

IF OperandSize is 64-bit

THEN OF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + OF;

ELSE OF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + OF;

FI;

Flags Affected

OF is updated based on result. CF, SF, ZF, AF and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);

unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD

If the LOCK prefix is used.

If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0)	For an illegal address in the SS segment.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

AESDEC—Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	RM	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDECLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

AESDEC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESDEC

STATE \leftarrow SRC1;
RoundKey \leftarrow SRC2;
STATE \leftarrow InvShiftRows(STATE);
STATE \leftarrow InvSubBytes(STATE);
STATE \leftarrow InvMixColumns(STATE);
DEST[127:0] \leftarrow STATE XOR RoundKey;
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC: `__m128i _mm_aesdec (__m128i, __m128i)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	RM	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**AESDECLAST**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST: `__m128i _mm_aesdeclast (__m128i, __m128i)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESENC—Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	RM	V/V	AES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a single round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**AESENC**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESENC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC: `__m128i _mm_aesenc (__m128i, __m128i)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	RM	V/V	AES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

AESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCLAST: `__m128i _mm_aesencast (__m128i, __m128i)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESIMC—Perform the AES InvMixColumn Transformation

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DB /r AESIMC xmm1, xmm2/m128	RM	V/V	AES	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.
VEX.128.66.0F38.WIG DB /r VAESIMC xmm1, xmm2/m128	RM	V/V	Both AES and AVX flags	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Perform the InvMixColumns transformation on the source operand and store the result in the destination operand. The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

Note: the AESIMC instruction should be applied to the expanded AES round keys (except for the first and last round key) in order to prepare them for decryption using the “Equivalent Inverse Cipher” (defined in FIPS 197).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

AESIMC

DEST[127:0] ← InvMixColumns(SRC);
DEST[VLMAX-1:128] (Unmodified)

VAESIMC

DEST[127:0] ← InvMixColumns(SRC);
DEST[VLMAX-1:128] ← 0;

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC: __m128i _mm_aesimc (__m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.vvvv ≠ 1111B.

AESKEYGENASSIST—AES Round Key Generation Assist

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A DF /r ib AESKEYGENASSIST xmm1, xmm2/m128, imm8	RMI	V/V	AES	Assist in AES round key generation using an 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F3A.WIG DF /r ib VAESKEYGENASSIST xmm1, xmm2/m128, imm8	RMI	V/V	Both AES and AVX flags	Assist in AES round key generation using 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

AESKEYGENASSIST

```

X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[VLMAX-1:128] (Unmodified)

```

VAESKEYGENASSIST

$X3[31:0] \leftarrow \text{SRC}[127:96];$
 $X2[31:0] \leftarrow \text{SRC}[95:64];$
 $X1[31:0] \leftarrow \text{SRC}[63:32];$
 $X0[31:0] \leftarrow \text{SRC}[31:0];$
 $\text{RCON}[31:0] \leftarrow \text{ZeroExtend}(\text{Imm8}[7:0]);$
 $\text{DEST}[31:0] \leftarrow \text{SubWord}(X1);$
 $\text{DEST}[63:32] \leftarrow \text{RotWord}(\text{SubWord}(X1)) \text{ XOR } \text{RCON};$
 $\text{DEST}[95:64] \leftarrow \text{SubWord}(X3);$
 $\text{DEST}[127:96] \leftarrow \text{RotWord}(\text{SubWord}(X3)) \text{ XOR } \text{RCON};$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0;$

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESKEYGENASSIST: `__m128i _mm_aeskeygenassist (__m128i, const int)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv \neq 1111B.

AND—Logical AND

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
24 <i>ib</i>	AND AL, <i>imm8</i>	I	Valid	Valid	AL AND <i>imm8</i> .
25 <i>iw</i>	AND AX, <i>imm16</i>	I	Valid	Valid	AX AND <i>imm16</i> .
25 <i>id</i>	AND EAX, <i>imm32</i>	I	Valid	Valid	EAX AND <i>imm32</i> .
REX.W + 25 <i>id</i>	AND RAX, <i>imm32</i>	I	Valid	N.E.	RAX AND <i>imm32</i> sign-extended to 64-bits.
80 /4 <i>ib</i>	AND <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> AND <i>imm8</i> .
REX + 80 /4 <i>ib</i>	AND <i>r/m8</i> [*] , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> AND <i>imm8</i> .
81 /4 <i>iw</i>	AND <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> AND <i>imm16</i> .
81 /4 <i>id</i>	AND <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> AND <i>imm32</i> .
REX.W + 81 /4 <i>id</i>	AND <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> AND <i>imm32</i> sign extended to 64-bits.
83 /4 <i>ib</i>	AND <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> AND <i>imm8</i> (sign-extended).
83 /4 <i>ib</i>	AND <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> AND <i>imm8</i> (sign-extended).
REX.W + 83 /4 <i>ib</i>	AND <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> AND <i>imm8</i> (sign-extended).
20 /r	AND <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> AND <i>r8</i> .
REX + 20 /r	AND <i>r/m8</i> [*] , <i>r8</i> [*]	MR	Valid	N.E.	<i>r/m64</i> AND <i>r8</i> (sign-extended).
21 /r	AND <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> AND <i>r16</i> .
21 /r	AND <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> AND <i>r32</i> .
REX.W + 21 /r	AND <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> AND <i>r32</i> .
22 /r	AND <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> AND <i>r/m8</i> .
REX + 22 /r	AND <i>r8</i> [*] , <i>r/m8</i> [*]	RM	Valid	N.E.	<i>r/m64</i> AND <i>r8</i> (sign-extended).
23 /r	AND <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> AND <i>r/m16</i> .
23 /r	AND <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> AND <i>r/m32</i> .
REX.W + 23 /r	AND <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> AND <i>r/m64</i> .

NOTES:

*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>r</i> , <i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA
MR	ModRM:r/m (<i>r</i> , <i>w</i>)	ModRM:reg (<i>r</i>)	NA	NA
MI	ModRM:r/m (<i>r</i> , <i>w</i>)	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX	<i>imm8</i>	NA	NA

Description

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← DEST AND SRC;

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ANDN — Logical AND NOT

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.OF38.W0 F2 /r ANDN r32a, r32b, r/m32	RVM	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.NDS.LZ.OF38.W1 F2 /r ANDN r64a, r64b, r/m64	RVM	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64a.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

DEST ← (NOT SRC1) bitwiseAND SRC2;

SF ← DEST[OperandSize - 1];

ZF ← (DEST = 0);

Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.

ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 54 /r ANDPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.OF.WIG 54 /r VANDPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.OF.WIG 54 /r VANDPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ANDPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]
 DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]
 DEST[VLMAX-1:128] (Unmodified)

VANDPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
 DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
 DEST[VLMAX-1:128] ← 0

VANDPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
 DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
 DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]
 DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

ANDPD: `__m128d _mm_and_pd(__m128d a, __m128d b)`

VANDPD: `__m256d _mm256_and_pd (__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 54 /r ANDPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Bitwise logical AND of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.OF.WIG 54 /r VANDPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 54 /r VANDPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ANDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VANDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VANDPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE AND } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE AND } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE AND } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE AND } \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE AND } \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE AND } \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE AND } \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE AND } \text{SRC2}[255:224]$.

Intel C/C++ Compiler Intrinsic Equivalent

ANDPS: `__m128 _mm_and_ps(__m128 a, __m128 b)`

VANDPS: `__m256 _mm256_and_ps (__m256 a, __m256 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 55 /r VANDNPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 55 /r VANDNPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND NOT of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**ANDNPD (128-bit Legacy SSE version)**

DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
 DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
 DEST[VLMAX-1:128] (Unmodified)

VANDNPD (VEX.128 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
 DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
 DEST[VLMAX-1:128] ← 0

VANDNPD (VEX.256 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
 DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
 DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
 DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPD: __m128d _mm_andnot_pd(__m128d a, __m128d b)

VANDNPD: __m256d _mm256_andnot_pd (__m256d a, __m256d b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 55 /r ANDNPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.OF.WIG 55 /r VANDNPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 55 /r VANDNPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Inverts the bits of the four packed single-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the four packed single-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ANDNPS (128-bit Legacy SSE version)

```
DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VANDNPS (VEX.128 encoded version)

```
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VANDNPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow (\text{NOT}(\text{SRC1}[31:0])) \text{ BITWISE AND } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow (\text{NOT}(\text{SRC1}[63:32])) \text{ BITWISE AND } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow (\text{NOT}(\text{SRC1}[95:64])) \text{ BITWISE AND } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow (\text{NOT}(\text{SRC1}[127:96])) \text{ BITWISE AND } \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow (\text{NOT}(\text{SRC1}[159:128])) \text{ BITWISE AND } \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow (\text{NOT}(\text{SRC1}[191:160])) \text{ BITWISE AND } \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow (\text{NOT}(\text{SRC1}[223:192])) \text{ BITWISE AND } \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow (\text{NOT}(\text{SRC1}[255:224])) \text{ BITWISE AND } \text{SRC2}[255:224]$.

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPS: `__m128 _mm_andnot_ps(__m128 a, __m128 b)`

VANDNPS: `__m256 _mm256_andnot_ps (__m256 a, __m256 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
63 /r	ARPL <i>r/m16, r16</i>	NP	N. E.	Valid	Adjust RPL of <i>r/m16</i> to not less than RPL of <i>r16</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

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

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the use of this instruction.

Operation

```

IF 64-BIT MODE
  THEN
    See MOVXSD;
  ELSE
    IF DEST[RPL] < SRC[RPL]
      THEN
        ZF ← 1;
        DEST[RPL] ← SRC[RPL];
      ELSE
        ZF ← 0;
    FI;
  FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	<p>The ARPL instruction is not recognized in real-address mode.</p> <p>If the LOCK prefix is used.</p>
-----	--

Virtual-8086 Mode Exceptions

#UD	<p>The ARPL instruction is not recognized in virtual-8086 mode.</p> <p>If the LOCK prefix is used.</p>
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Not applicable.

BLENDPD — Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 0D /r ib BLENDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed DP-FP values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0D /r ib VBLENDPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0D /r ib VBLENDPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

Description

Double-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [3:0] determine whether the corresponding double-precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is "1", then the double-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

BLENDPD (128-bit Legacy SSE version)

```
IF (IMM8[0] = 0) THEN DEST[63:0] ← DEST[63:0]
  ELSE DEST [63:0] ← SRC[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
  ELSE DEST [127:64] ← SRC[127:64] FI
DEST[VLMAX-1:128] (Unmodified)
```

VBLENDPD (VEX.128 encoded version)

```
IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
  ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[VLMAX-1:128] ← 0
```

VBLENDPD (VEX.256 encoded version)

```

IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
IF (IMM8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST [191:128] ← SRC2[191:128] FI
IF (IMM8[3] = 0) THEN DEST[255:192] ← SRC1[255:192]
    ELSE DEST [255:192] ← SRC2[255:192] FI

```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPD: __m128d _mm_blend_pd (__m128d v1, __m128d v2, const int mask);

VBLENDPD: __m256d _mm256_blend_pd (__m256d a, __m256d b, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

BEXTR — Bit Field Extract

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS ¹ .LZ.OF38.W0 F7 /r BEXTR r32a, r/m32, r32b	RMV	V/V	BMI1	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.NDS ¹ .LZ.OF38.W1 F7 /r BEXTR r64a, r/m64, r64b	RMV	V/N.E.	BMI1	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a

NOTES:

1. ModRM:r/m is used to encode the first source operand (second operand) and VEX.vvvv encodes the second source operand (third operand).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize - 1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
START ← SRC2[7:0];
LEN ← SRC2[15:8];
TEMP ← ZERO_EXTEND_TO_512 (SRC1 );
DEST ← ZERO_EXTEND(TEMP[START+LEN -1: START]);
ZF ← (DEST = 0);
```

Flags Affected

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

BEXTR: unsigned __int32 _bextr_u32(unsigned __int32 src, unsigned __int32 start, unsigned __int32 len);

BEXTR: unsigned __int64 _bextr_u64(unsigned __int64 src, unsigned __int32 start, unsigned __int32 len);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

BLENDPS — Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 0C /r ib BLENDPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0C /r ib VBLENDPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0C /r ib VBLENDPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Packed single-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [7:0] determine whether the corresponding single precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is "1", then the single-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

BLENDPS (128-bit Legacy SSE version)

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[VLMAX-1:128] (Unmodified)

```

VBLENDPS (VEX.128 encoded version)

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[VLMAX-1:128] ← 0

```

VBLENDPS (VEX.256 encoded version)

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
IF (IMM8[4] = 0) THEN DEST[159:128] ← SRC1[159:128]
    ELSE DEST [159:128] ← SRC2[159:128] FI
IF (IMM8[5] = 0) THEN DEST[191:160] ← SRC1[191:160]
    ELSE DEST [191:160] ← SRC2[191:160] FI
IF (IMM8[6] = 0) THEN DEST[223:192] ← SRC1[223:192]
    ELSE DEST [223:192] ← SRC2[223:192] FI
IF (IMM8[7] = 0) THEN DEST[255:224] ← SRC1[255:224]
    ELSE DEST [255:224] ← SRC2[255:224] FI.

```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPS: __m128 _mm_blend_ps (__m128 v1, __m128 v2, const int mask);

VBLENDPS: __m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RM0	V/V	SSE4_1	Select packed DP FP values from <i>xmm1</i> and <i>xmm2</i> from mask specified in <i>XMM0</i> and store the values in <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the mask operand, <i>ymm4</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy each quadword data element of double-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each quadword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding quadword element in the second source operand, If a mask bit is "1"; or
- the corresponding quadword element in the first source operand, If a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPD is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (VLMAX-1:128) of the corresponding YMM register (destination register) are zeroed.

VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPD permits the mask to be any XMM or YMM register. In contrast, BLENDVPD treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

Operation

BLENDVPD (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[63] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[VLMAX-1:128] (Unmodified)
```

VBLENDVPD (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[VLMAX-1:128] ← 0
```

VBLENDVPD (VEX.256 encoded version)

```
MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
IF (MASK[191] = 0) THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST [191:128] ← SRC2[191:128] FI
IF (MASK[255] = 0) THEN DEST[255:192] ← SRC1[255:192]
    ELSE DEST [255:192] ← SRC2[255:192] FI
```

Intel C/C++ Compiler Intrinsic Equivalent

```
BLENDVPD:   __m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3);
VBLENDVPD:  __m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);
VBLENDVPD:  __m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RMO	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the specified mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the specified mask register, <i>ymm4</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMO	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy each dword data element of single-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each dword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding dword element in the second source operand, If a mask bit is "1"; or
- the corresponding dword element in the first source operand, If a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPS is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (VLMAX-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPS permits the mask to be any XMM or YMM register. In contrast, BLENDVPS treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

Operation

BLENDVPS (128-bit Legacy SSE version)

```

MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[VLMAX-1:128] (Unmodified)

```

VBLENDVPS (VEX.128 encoded version)

```

MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[VLMAX-1:128] ← 0

```

VBLENDVPS (VEX.256 encoded version)

```

MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
IF (MASK[159] = 0) THEN DEST[159:128] ← SRC1[159:128]
    ELSE DEST [159:128] ← SRC2[159:128] FI
IF (MASK[191] = 0) THEN DEST[191:160] ← SRC1[191:160]
    ELSE DEST [191:160] ← SRC2[191:160] FI
IF (MASK[223] = 0) THEN DEST[223:192] ← SRC1[223:192]
    ELSE DEST [223:192] ← SRC2[223:192] FI
IF (MASK[255] = 0) THEN DEST[255:224] ← SRC1[255:224]
    ELSE DEST [255:224] ← SRC2[255:224] FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```

BLENDVPS:    __m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3);
VBLENDVPS:   __m128 _mm_blendv_ps (__m128 a, __m128 b, __m128 mask);
VBLENDVPS:   __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.W = 1.

BLSI — Extract Lowest Set Isolated Bit

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.OF38.W0 F3 /3 BLSI r32, r/m32	VM	V/V	BMI1	Extract lowest set bit from r/m32 and set that bit in r32.
VEX.NDD.LZ.OF38.W1 F3 /3 BLSI r64, r/m64	VM	V/N.E.	BMI1	Extract lowest set bit from r/m64, and set that bit in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (-SRC) bitwiseAND (SRC);
SF ← temp[OperandSize - 1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 0;
ELSE
    CF ← 1;
FI
DEST ← temp;
```

Flags Affected

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```
BLSI:    unsigned __int32 _bsi_u32(unsigned __int32 src);
BLSI:    unsigned __int64 _bsi_u64(unsigned __int64 src);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

BLSMSK — Get Mask Up to Lowest Set Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.OF38.W0 F3 /2 BLSMSK r32, r/m32	VM	V/V	BMI1	Set all lower bits in r32 to “1” starting from bit 0 to lowest set bit in r/m32.
VEX.NDD.LZ.OF38.W1 F3 /2 BLSMSK r64, r/m64	VM	V/N.E.	BMI1	Set all lower bits in r64 to “1” starting from bit 0 to lowest set bit in r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Sets all the lower bits of the destination operand to “1” up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) XOR (SRC) ;
SF ← temp[OperandSize - 1];
ZF ← 0;
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

SF is updated based on the result. CF is set if the source if zero. ZF and OF flags are cleared. AF and PF flag are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSMSK: unsigned __int32 _blsmask_u32(unsigned __int32 src);

BLSMSK: unsigned __int64 _blsmask_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.

BLSR — Reset Lowest Set Bit

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.OF38.W0 F3 /1 BLSR r32, r/m32	VM	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.NDD.LZ.OF38.W1 F3 /1 BLSR r64, r/m64	VM	V/N.E.	BMI1	Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) bitwiseAND ( SRC );
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSR: unsigned __int32 _blsr_u32(unsigned __int32 src);

BLSR: unsigned __int64 _blsr_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
62 /r	BOUND <i>r16</i> , <i>m16&16</i>	RM	Invalid	Valid	Check if <i>r16</i> (array index) is within bounds specified by <i>m16&16</i> .
62 /r	BOUND <i>r32</i> , <i>m32&32</i>	RM	Invalid	Valid	Check if <i>r32</i> (array index) is within bounds specified by <i>m32&32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>r</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

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

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

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64bit Mode
  THEN
    #UD;
  ELSE
    IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound)
      (* Below lower bound or above upper bound *)
      THEN #BR; FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	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

#BR If the bounds test fails.
 #UD If second operand is not a memory location.
 If the LOCK prefix is used.
 #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.

Virtual-8086 Mode Exceptions

#BR If the bounds test fails.
 #UD If second operand is not a memory location.
 If the LOCK prefix is used.
 #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
 #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.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

BSF—Bit Scan Forward

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF BC /r	BSF <i>r16, r/m16</i>	RM	Valid	Valid	Bit scan forward on <i>r/m16</i> .
OF BC /r	BSF <i>r32, r/m32</i>	RM	Valid	Valid	Bit scan forward on <i>r/m32</i> .
REX.W + OF BC /r	BSF <i>r64, r/m64</i>	RM	Valid	N.E.	Bit scan forward on <i>r/m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← 0;
        WHILE Bit(SRC, temp) = 0
            DO
                temp ← temp + 1;
            OD;
        DEST ← temp;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

BSR—Bit Scan Reverse

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF BD /r	BSR <i>r16, r/m16</i>	RM	Valid	Valid	Bit scan reverse on <i>r/m16</i> .
OF BD /r	BSR <i>r32, r/m32</i>	RM	Valid	Valid	Bit scan reverse on <i>r/m32</i> .
REX.W + OF BD /r	BSR <i>r64, r/m64</i>	RM	Valid	N.E.	Bit scan reverse on <i>r/m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← OperandSize - 1;
        WHILE Bit(SRC, temp) = 0
            DO
                temp ← temp - 1;
            OD;
        DEST ← temp;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

BSWAP—Byte Swap

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF C8+rd	BSWAP r32	0	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + OF C8+rd	BSWAP r64	0	Valid	N.E.	Reverses the byte order of a 64-bit register.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
0	opcode + rd (r, w)	NA	NA	NA

Description

Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Legacy Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

Operation

```
TEMP ← DEST
IF 64-bit mode AND OperandSize = 64
  THEN
    DEST[7:0] ← TEMP[63:56];
    DEST[15:8] ← TEMP[55:48];
    DEST[23:16] ← TEMP[47:40];
    DEST[31:24] ← TEMP[39:32];
    DEST[39:32] ← TEMP[31:24];
    DEST[47:40] ← TEMP[23:16];
    DEST[55:48] ← TEMP[15:8];
    DEST[63:56] ← TEMP[7:0];
  ELSE
    DEST[7:0] ← TEMP[31:24];
    DEST[15:8] ← TEMP[23:16];
    DEST[23:16] ← TEMP[15:8];
    DEST[31:24] ← TEMP[7:0];
```

FI;

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

BT—Bit Test

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF A3 /r	BT <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag.
OF A3 /r	BT <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + OF A3 /r	BT <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag.
OF BA /4 <i>ib</i>	BT <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag.
OF BA /4 <i>ib</i>	BT <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag.
REX.W + OF BA /4 <i>ib</i>	BT <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

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

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

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

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

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

CF ← Bit(BitBase, BitOffset);

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

BTC—Bit Test and Complement

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF BB /r	BTC <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
OF BB /r	BTC <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BB /r	BTC <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BA /7 <i>ib</i>	BTC <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and complement.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

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

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

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

$CF \leftarrow \text{Bit}(\text{BitBase}, \text{BitOffset});$

$\text{Bit}(\text{BitBase}, \text{BitOffset}) \leftarrow \text{NOT } \text{Bit}(\text{BitBase}, \text{BitOffset});$

Flags Affected

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

Protected Mode Exceptions

- #GP(0) If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTR—Bit Test and Reset

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF B3 /r	BTR <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and clear.
OF B3 /r	BTR <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF B3 /r	BTR <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and clear.
OF BA /6 <i>ib</i>	BTR <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and clear.
OF BA /6 <i>ib</i>	BTR <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF BA /6 <i>ib</i>	BTR <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and clear.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (<i>r, w</i>)	ModRM:reg (<i>r</i>)	NA	NA
MI	ModRM:r/m (<i>r, w</i>)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

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

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

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

CF ← Bit(BitBase, BitOffset);
 Bit(BitBase, BitOffset) ← 0;

Flags Affected

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

Protected Mode Exceptions

- #GP(0) If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTS—Bit Test and Set

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF AB /r	BTS <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
OF AB /r	BTS <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF AB /r	BTS <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF BA /5 <i>ib</i>	BTS <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and set.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (<i>r, w</i>)	ModRM:reg (<i>r</i>)	NA	NA
MI	ModRM:r/m (<i>r, w</i>)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

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

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

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← 1;

Flags Affected

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

Protected Mode Exceptions

- #GP(0) If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 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.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BZHI — Zero High Bits Starting with Specified Bit Position

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS ¹ .LZ.OF38.W0 F5 /r BZHI r32a, r/m32, r32b	RMV	V/V	BMI2	Zero bits in r/m32 starting with the position in r32b, write result to r32a.
VEX.NDS ¹ .LZ.OF38.W1 F5 /r BZHI r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Zero bits in r/m64 starting with the position in r64b, write result to r64a.

NOTES:

1. ModRM:r/m is used to encode the first source operand (second operand) and VEX.vvvv encodes the second source operand (third operand).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize - 1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize - 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```

N ← SRC2[7:0]
DEST ← SRC1
IF (N < OperandSize)
    DEST[OperandSize-1:N] ← 0
FI
IF (N > OperandSize - 1)
    CF ← 1
ELSE
    CF ← 0
FI

```

Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BZHI: unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);

BZHI: unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

CALL—Call Procedure

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
E8 <i>cw</i>	CALL <i>rel16</i>	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 <i>cd</i>	CALL <i>rel32</i>	M	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL <i>r/m16</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m16</i> .
FF /2	CALL <i>r/m32</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m32</i> .
FF /2	CALL <i>r/m64</i>	M	Valid	N.E.	Call near, absolute indirect, address given in <i>r/m64</i> .
9A <i>cd</i>	CALL <i>ptr16:16</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
9A <i>cp</i>	CALL <i>ptr16:32</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL <i>m16:16</i>	M	Valid	Valid	Call far, absolute indirect address given in <i>m16:16</i> . In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL <i>m16:32</i>	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL <i>m16:64</i>	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

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

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

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

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, “Task Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

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

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32*, or *r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real- address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

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

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

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

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

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

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16: 16* or *ptr16: 32*) or indirectly with a memory location (*m16: 16* or *m16: 32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedure's stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute a task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

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

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

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

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16: 16* or *ptr16: 32*) or indirectly with a memory location (*m16: 16* or *m16: 32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This is specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the

offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Operation

```

IF near call
  THEN IF near relative call
    THEN
      IF OperandSize = 64
        THEN
          tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
          tempRIP ← RIP + tempDEST;
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); FI;
          Push(RIP);
          RIP ← tempRIP;
        FI;
      IF OperandSize = 32
        THEN
          tempEIP ← EIP + DEST; (* DEST is rel32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); FI;
          Push(EIP);
          EIP ← tempEIP;
        FI;
      IF OperandSize = 16
        THEN
          tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 2-byte return address
            THEN #SS(0); FI;
          Push(IP);
          EIP ← tempEIP;
        FI;
      ELSE (* Near absolute call *)
        IF OperandSize = 64
          THEN
            tempRIP ← DEST; (* DEST is r/m64 *)
            IF stack not large enough for a 8-byte return address
              THEN #SS(0); FI;
            Push(RIP);
            RIP ← tempRIP;
          FI;
        IF OperandSize = 32
          THEN
            tempEIP ← DEST; (* DEST is r/m32 *)
            IF tempEIP is not within code segment limit THEN #GP(0); FI;
            IF stack not large enough for a 4-byte return address
              THEN #SS(0); FI;
            Push(EIP);
            EIP ← tempEIP;
          FI;

```



```

    FI;
    IF OperandSize = 16
        THEN
            tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
            IF tempEIP is not within code segment limit THEN #GP(0); FI;
            IF stack not large enough for a 2-byte return address
                THEN #SS(0); FI;
            Push(IP);
            EIP ← tempEIP;
        FI;
    FI;rel/abs
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        IF OperandSize = 32
            THEN
                IF stack not large enough for a 6-byte return address
                    THEN #SS(0); FI;
                IF DEST[31:16] is not zero THEN #GP(0); FI;
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(CS);
                Push(IP);
                CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
            FI;
        FI;
    FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
    THEN
        IF segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector); FI;
        Read type and access rights of selected segment descriptor;
        IF IA32_EFER.LMA = 0
            THEN
                IF segment type is not a conforming or nonconforming code segment, call
                    gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment or
                    64-bit call gate,
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
        GO TO CONFORMING-CODE-SEGMENT;
        GO TO NONCONFORMING-CODE-SEGMENT;

```

```

GO TO CALL-GATE;
GO TO TASK-GATE;
GO TO TASK-STATE-SEGMENT;

```

```
FI;
```

CONFORMING-CODE-SEGMENT:

```

IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF DPL > CPL
    THEN #GP(new code segment selector); FI;
IF segment not present
    THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
    THEN #SS(0); FI;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN
        tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
ELSE
    IF OperandSize = 16
        THEN
            Push(CS);
            Push(IP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE (* OperandSize = 64 *)
            Push(CS); (* Padded with 48 high-order bits *)
            Push(RIP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            RIP ← tempEIP;

```

```
FI;
```

```
FI;
```

```
END;
```

NONCONFORMING-CODE-SEGMENT:

```

IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;

```

```

IF (RPL > CPL) or (DPL ≠ CPL)
    THEN #GP(new code segment selector); FI;
IF segment not present
    THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
    THEN #SS(0); FI;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
                Push(RIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                RIP ← tempEIP;
        FI;
    FI;
FI;
END;

```

CALL-GATE:

```

IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); FI;
IF call gate not present
    THEN #NP(call-gate selector); FI;
IF call-gate code-segment selector is NULL
    THEN #GP(0); FI;
IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); FI;
Read call-gate code-segment descriptor;
IF call-gate code-segment descriptor does not indicate a code segment
or call-gate code-segment descriptor DPL > CPL

```

```

    THEN #GP(call-gate code-segment selector); FI;
IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
    THEN #GP(call-gate code-segment selector); FI;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); FI;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit
        THEN
            TSSstackAddress ← (new code-segment DPL * 8) + 4;
            IF (TSSstackAddress + 5) > current TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
            NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
        ELSE
            IF current TSS is 16-bit
                THEN
                    TSSstackAddress ← (new code-segment DPL * 4) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
                ELSE (* current TSS is 64-bit *)
                    TSSstackAddress ← (new code-segment DPL * 8) + 4;
                    IF (TSSstackAddress + 7) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                    NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            FI;
        FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read new code-segment descriptor and new stack-segment descriptor;
    IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
writable data segment)
        THEN #TS(NewSS); FI;
    IF IA32_EFER.LMA = 0 and new stack segment not present
        THEN #SS(NewSS); FI;
    IF CallGateSize = 32
        THEN
            IF new stack does not have room for parameters plus 16 bytes
                THEN #SS(NewSS); FI;
            IF CallGate(InstructionPointer) not within new code-segment limit
                THEN #GP(0); FI;
            SS ← newSS; (* Segment descriptor information also loaded *)
            ESP ← newESP;
            CS:EIP ← CallGate(CS:InstructionPointer);

```

```

(* Segment descriptor information also loaded *)
Push(oldSS:oldESP); (* From calling procedure *)
temp ← parameter count from call gate, masked to 5 bits;
Push(parameters from calling procedure's stack, temp)
Push(oldCS:oldEIP); (* Return address to calling procedure *)
ELSE
  IF CallGateSize = 16
    THEN
      IF new stack does not have room for parameters plus 8 bytes
        THEN #SS(NewSS); FI;
      IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
        THEN #GP(0); FI;
      SS ← newSS; (* Segment descriptor information also loaded *)
      ESP ← newESP;
      CS:IP ← CallGate(CS:InstructionPointer);
      (* Segment descriptor information also loaded *)
      Push(oldSS:oldESP); (* From calling procedure *)
      temp ← parameter count from call gate, masked to 5 bits;
      Push(parameters from calling procedure's stack, temp)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE (* CallGateSize = 64 *)
      IF pushing 32 bytes on the stack would use a non-canonical address
        THEN #SS(NewSS); FI;
      IF (CallGate(InstructionPointer) is non-canonical)
        THEN #GP(0); FI;
      SS ← NewSS; (* NewSS is NULL)
      RSP ← NewESP;
      CS:IP ← CallGate(CS:InstructionPointer);
      (* Segment descriptor information also loaded *)
      Push(oldSS:oldESP); (* From calling procedure *)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
    FI;
  FI;
CPL ← CodeSegment(DPL)
CS(RPL) ← CPL
END;

```

SAME-PRIVILEGE:

```

  IF CallGateSize = 32
    THEN
      IF stack does not have room for 8 bytes
        THEN #SS(0); FI;
      IF CallGate(InstructionPointer) not within code segment limit
        THEN #GP(0); FI;
      CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
      If CallGateSize = 16
        THEN
          IF stack does not have room for 4 bytes
            THEN #SS(0); FI;
          IF CallGate(InstructionPointer) not within code segment limit
            THEN #GP(0); FI;
          CS:IP ← CallGate(CS:instruction pointer);

```

```

        (* Segment descriptor information also loaded *)
        Push(oldCS:oldIP); (* Return address to calling procedure *)
    ELSE (* CallGateSize = 64)
        IF pushing 16 bytes on the stack touches non-canonical addresses
            THEN #SS(0); FI;
        IF RIP non-canonical
            THEN #GP(0); FI;
        CS:IP ← CallGate(CS:instruction pointer);
        (* Segment descriptor information also loaded *)
        Push(oldCS:oldIP); (* Return address to calling procedure *)
    FI;
FI;
CS(RPL) ← CPL
END;

```

TASK-GATE:

```

    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        or index not within GDT limits
            THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

TASK-STATE-SEGMENT:

```

    IF TSS DPL < CPL or RPL
        or TSS descriptor indicates TSS not available
            THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0) If the target offset in destination operand is beyond the new code segment limit.
 If the segment selector in the destination operand is NULL.
 If the code segment selector in the gate is NULL.

	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If a code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.</p>
#SS(selector)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.</p> <p>If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.</p> <p>If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.</p>
#NP(selector)	<p>If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.</p>
#TS(selector)	<p>If the new stack segment selector and ESP are beyond the end of the TSS.</p> <p>If the new stack segment selector is NULL.</p> <p>If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.</p> <p>If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.</p> <p>If the new stack segment is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</p>
#UD	<p>If the LOCK prefix is used.</p>

Real-Address Mode Exceptions

#GP	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
#UD	<p>If the LOCK prefix is used.</p>

Virtual-8086 Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
--------	--

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)	<p>If a memory address is non-canonical.</p> <p>If target offset in destination operand is non-canonical.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the 64-bit gate is NULL.</p>
#GP(selector)	<p>If code segment or 64-bit call gate is outside descriptor table limits.</p> <p>If code segment or 64-bit call gate overlaps non-canonical space.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.</p> <p>If the upper type field of a 64-bit call gate is not 0x0.</p> <p>If the segment selector from a 64-bit call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.</p> <p>If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.</p> <p>If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.</p>
#SS(0)	<p>If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.</p> <p>If a memory operand effective address is outside the SS segment limit.</p> <p>If the stack address is in a non-canonical form.</p>
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	<p>(64-bit mode only) If a far call is direct to an absolute address in memory.</p> <p>If the LOCK prefix is used.</p>
#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.

CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
98	CBW	NP	Valid	Valid	AX ← sign-extend of AL.
98	CWDE	NP	Valid	Valid	EAX ← sign-extend of AX.
REX.W + 98	CDQE	NP	Valid	N.E.	RAX ← sign-extend of EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

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

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

Operation

```
IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX ← SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX ← SignExtend(AX); FI;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX ← SignExtend(EAX);
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLAC—Clear AC Flag in EFLAGS Register

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF 01 CA	CLAC	NP	Valid	Valid	Clear the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

Operation

EFLAGS.AC ← 0;

Flags Affected

AC cleared. Other flags are unaffected.

Protected Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD
 If the LOCK prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

#UD The CLAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

CLC—Clear Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F8	CLC	NP	Valid	Valid	Clear CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

CF ← 0;

Flags Affected

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

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLD—Clear Direction Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
FC	CLD	NP	Valid	Valid	Clear DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

Operation

$DF \leftarrow 0$;

Flags Affected

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

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLFLUSH—Flush Cache Line

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF AE /7	CLFLUSH <i>m8</i>	M	Valid	Valid	Flushes cache line containing <i>m8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see “CPUID—CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH/h instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH/h instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to ensure that previous stores are included in the write-back.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Flush_Cache_Line(SRC);
```

Intel C/C++ Compiler Intrinsic Equivalents

```
CLFLUSH: void _mm_clflush(void const *p)
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0. If the LOCK prefix is used.

If instruction prefix is 66H, F2H or F3H.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0. If the LOCK prefix is used. If instruction prefix is 66H, F2H or F3H.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
-----------------	-------------------

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0. If the LOCK prefix is used. If instruction prefix is 66H, F2H or F3H.

CLI — Clear Interrupt Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
FA	CLI	NP	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

If protected-mode virtual interrupts are not enabled, CLI clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected. Table 3-6 indicates the action of the CLI instruction depending on the processor operating mode and the CPL/IOPL of the running program or procedure.

Operation is the same in all modes.

Table 3-6. Decision Table for CLI Results

PE	VM	IOPL	CPL	PVI	VIP	VME	CLI Result
0	X	X	X	X	X	X	IF = 0
1	0	≥ CPL	X	X	X	X	IF = 0
1	0	< CPL	3	1	X	X	VIF = 0
1	0	< CPL	< 3	X	X	X	GP Fault
1	0	< CPL	X	0	X	X	GP Fault
1	1	3	X	X	X	X	IF = 0
1	1	< 3	X	X	X	1	VIF = 0
1	1	< 3	X	X	X	0	GP Fault

NOTES:

* X = This setting has no impact.

Operation

```

IF PE = 0
  THEN
    IF ← 0; (* Reset Interrupt Flag *)
  ELSE
    IF VM = 0;
      THEN
        IF IOPL ≥ CPL
          THEN
            IF ← 0; (* Reset Interrupt Flag *)
          ELSE
            IF ((IOPL < CPL) and (CPL = 3) and (PVI = 1))
              THEN
                VIF ← 0; (* Reset Virtual Interrupt Flag *)
              ELSE

```

```

                                #GP(0);
                                FI;
                                FI;
                                ELSE (* VM = 1 *)
                                IF IOPL = 3
                                THEN
                                IF ← 0; (* Reset Interrupt Flag *)
                                ELSE
                                IF (IOPL < 3) AND (VME = 1)
                                THEN
                                VIF ← 0; (* Reset Virtual Interrupt Flag *)
                                ELSE
                                #GP(0);
                                FI;
                                FI;
                                FI;
                                FI;
                                FI;

```

Flags Affected

If protected-mode virtual interrupts are not enabled, IF is set to 0 if the CPL is equal to or less than the IOPL; otherwise, it is not affected. Other flags are unaffected.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

Virtual-8086 Mode Exceptions

#GP(0)	If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
#UD	If the LOCK prefix is used.

CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF 06	CLTS	NP	Valid	Valid	Clears TS flag in CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 25, “VMX Non-Root Operation,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

CR0.TS[bit 3] ← 0;

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) CLTS is not recognized in virtual-8086 mode.
 #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the CPL is greater than 0.
 #UD If the LOCK prefix is used.

CMC—Complement Carry Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
F5	CMC	NP	Valid	Valid	Complement CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

Operation

$EFLAGS.CF[bit\ 0] \leftarrow NOT\ EFLAGS.CF[bit\ 0];$

Flags Affected

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

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CMOVcc—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 47 /r	CMOVA <i>r16, r/m16</i>	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
OF 47 /r	CMOVA <i>r32, r/m32</i>	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVA <i>r64, r/m64</i>	RM	Valid	N.E.	Move if above (CF=0 and ZF=0).
OF 43 /r	CMOVAE <i>r16, r/m16</i>	RM	Valid	Valid	Move if above or equal (CF=0).
OF 43 /r	CMOVAE <i>r32, r/m32</i>	RM	Valid	Valid	Move if above or equal (CF=0).
REX.W + OF 43 /r	CMOVAE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if above or equal (CF=0).
OF 42 /r	CMOVB <i>r16, r/m16</i>	RM	Valid	Valid	Move if below (CF=1).
OF 42 /r	CMOVB <i>r32, r/m32</i>	RM	Valid	Valid	Move if below (CF=1).
REX.W + OF 42 /r	CMOVB <i>r64, r/m64</i>	RM	Valid	N.E.	Move if below (CF=1).
OF 46 /r	CMOVBE <i>r16, r/m16</i>	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
OF 46 /r	CMOVBE <i>r32, r/m32</i>	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + OF 46 /r	CMOVBE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).
OF 42 /r	CMOVC <i>r16, r/m16</i>	RM	Valid	Valid	Move if carry (CF=1).
OF 42 /r	CMOVC <i>r32, r/m32</i>	RM	Valid	Valid	Move if carry (CF=1).
REX.W + OF 42 /r	CMOVC <i>r64, r/m64</i>	RM	Valid	N.E.	Move if carry (CF=1).
OF 44 /r	CMOVE <i>r16, r/m16</i>	RM	Valid	Valid	Move if equal (ZF=1).
OF 44 /r	CMOVE <i>r32, r/m32</i>	RM	Valid	Valid	Move if equal (ZF=1).
REX.W + OF 44 /r	CMOVE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if equal (ZF=1).
OF 4F /r	CMOVG <i>r16, r/m16</i>	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
OF 4F /r	CMOVG <i>r32, r/m32</i>	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
REX.W + OF 4F /r	CMOVG <i>r64, r/m64</i>	RM	V/N.E.	NA	Move if greater (ZF=0 and SF=OF).
OF 4D /r	CMOVGE <i>r16, r/m16</i>	RM	Valid	Valid	Move if greater or equal (SF=OF).
OF 4D /r	CMOVGE <i>r32, r/m32</i>	RM	Valid	Valid	Move if greater or equal (SF=OF).
REX.W + OF 4D /r	CMOVGE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if greater or equal (SF=OF).
OF 4C /r	CMOVL <i>r16, r/m16</i>	RM	Valid	Valid	Move if less (SF≠ OF).
OF 4C /r	CMOVL <i>r32, r/m32</i>	RM	Valid	Valid	Move if less (SF≠ OF).
REX.W + OF 4C /r	CMOVL <i>r64, r/m64</i>	RM	Valid	N.E.	Move if less (SF≠ OF).
OF 4E /r	CMOVLE <i>r16, r/m16</i>	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
OF 4E /r	CMOVLE <i>r32, r/m32</i>	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
REX.W + OF 4E /r	CMOVLE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ OF).
OF 46 /r	CMOVNA <i>r16, r/m16</i>	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
OF 46 /r	CMOVNA <i>r32, r/m32</i>	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + OF 46 /r	CMOVNA <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not above (CF=1 or ZF=1).
OF 42 /r	CMOVNAE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not above or equal (CF=1).
OF 42 /r	CMOVNAE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not above or equal (CF=1).
REX.W + OF 42 /r	CMOVNAE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not above or equal (CF=1).
OF 43 /r	CMOVNB <i>r16, r/m16</i>	RM	Valid	Valid	Move if not below (CF=0).
OF 43 /r	CMOVNB <i>r32, r/m32</i>	RM	Valid	Valid	Move if not below (CF=0).
REX.W + OF 43 /r	CMOVNB <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not below (CF=0).
OF 47 /r	CMOVNBE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 47 /r	CMOVNBE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVNBE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
OF 43 /r	CMOVNC <i>r16, r/m16</i>	RM	Valid	Valid	Move if not carry (CF=0).
OF 43 /r	CMOVNC <i>r32, r/m32</i>	RM	Valid	Valid	Move if not carry (CF=0).
REX.W + OF 43 /r	CMOVNC <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not carry (CF=0).
OF 45 /r	CMOVNE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not equal (ZF=0).
OF 45 /r	CMOVNE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not equal (ZF=0).
REX.W + OF 45 /r	CMOVNE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not equal (ZF=0).
OF 4E /r	CMOVNG <i>r16, r/m16</i>	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠OF).
OF 4E /r	CMOVNG <i>r32, r/m32</i>	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠OF).
REX.W + OF 4E /r	CMOVNG <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not greater (ZF=1 or SF≠OF).
OF 4C /r	CMOVNGE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not greater or equal (SF≠OF).
OF 4C /r	CMOVNGE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not greater or equal (SF≠OF).
REX.W + OF 4C /r	CMOVNGE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not greater or equal (SF≠OF).
OF 4D /r	CMOVNL <i>r16, r/m16</i>	RM	Valid	Valid	Move if not less (SF=OF).
OF 4D /r	CMOVNL <i>r32, r/m32</i>	RM	Valid	Valid	Move if not less (SF=OF).
REX.W + OF 4D /r	CMOVNL <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not less (SF=OF).
OF 4F /r	CMOVNLE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
OF 4F /r	CMOVNLE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + OF 4F /r	CMOVNLE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
OF 41 /r	CMOVNO <i>r16, r/m16</i>	RM	Valid	Valid	Move if not overflow (OF=0).
OF 41 /r	CMOVNO <i>r32, r/m32</i>	RM	Valid	Valid	Move if not overflow (OF=0).
REX.W + OF 41 /r	CMOVNO <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not overflow (OF=0).
OF 4B /r	CMOVNP <i>r16, r/m16</i>	RM	Valid	Valid	Move if not parity (PF=0).
OF 4B /r	CMOVNP <i>r32, r/m32</i>	RM	Valid	Valid	Move if not parity (PF=0).
REX.W + OF 4B /r	CMOVNP <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not parity (PF=0).
OF 49 /r	CMOVNS <i>r16, r/m16</i>	RM	Valid	Valid	Move if not sign (SF=0).
OF 49 /r	CMOVNS <i>r32, r/m32</i>	RM	Valid	Valid	Move if not sign (SF=0).
REX.W + OF 49 /r	CMOVNS <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not sign (SF=0).
OF 45 /r	CMOVNZ <i>r16, r/m16</i>	RM	Valid	Valid	Move if not zero (ZF=0).
OF 45 /r	CMOVNZ <i>r32, r/m32</i>	RM	Valid	Valid	Move if not zero (ZF=0).
REX.W + OF 45 /r	CMOVNZ <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not zero (ZF=0).
OF 40 /r	CMOVO <i>r16, r/m16</i>	RM	Valid	Valid	Move if overflow (OF=1).
OF 40 /r	CMOVO <i>r32, r/m32</i>	RM	Valid	Valid	Move if overflow (OF=1).
REX.W + OF 40 /r	CMOVO <i>r64, r/m64</i>	RM	Valid	N.E.	Move if overflow (OF=1).
OF 4A /r	CMOVPP <i>r16, r/m16</i>	RM	Valid	Valid	Move if parity (PF=1).
OF 4A /r	CMOVPP <i>r32, r/m32</i>	RM	Valid	Valid	Move if parity (PF=1).
REX.W + OF 4A /r	CMOVPP <i>r64, r/m64</i>	RM	Valid	N.E.	Move if parity (PF=1).
OF 4A /r	CMOVPE <i>r16, r/m16</i>	RM	Valid	Valid	Move if parity even (PF=1).
OF 4A /r	CMOVPE <i>r32, r/m32</i>	RM	Valid	Valid	Move if parity even (PF=1).
REX.W + OF 4A /r	CMOVPE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if parity even (PF=1).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 4B /r	CMOVPO <i>r16, r/m16</i>	RM	Valid	Valid	Move if parity odd (PF=0).
OF 4B /r	CMOVPO <i>r32, r/m32</i>	RM	Valid	Valid	Move if parity odd (PF=0).
REX.W + OF 4B /r	CMOVPO <i>r64, r/m64</i>	RM	Valid	N.E.	Move if parity odd (PF=0).
OF 48 /r	CMOVVS <i>r16, r/m16</i>	RM	Valid	Valid	Move if sign (SF=1).
OF 48 /r	CMOVVS <i>r32, r/m32</i>	RM	Valid	Valid	Move if sign (SF=1).
REX.W + OF 48 /r	CMOVVS <i>r64, r/m64</i>	RM	Valid	N.E.	Move if sign (SF=1).
OF 44 /r	CMOVZ <i>r16, r/m16</i>	RM	Valid	Valid	Move if zero (ZF=1).
OF 44 /r	CMOVZ <i>r32, r/m32</i>	RM	Valid	Valid	Move if zero (ZF=1).
REX.W + OF 44 /r	CMOVZ <i>r64, r/m64</i>	RM	Valid	N.E.	Move if zero (ZF=1).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>r, w</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

The CMOV_{cc} instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV_{cc} instruction.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOV_{cc} mnemonic is given in the description column of the above table. 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 CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOV_{cc} instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOV_{cc} instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

temp ← SRC

IF condition TRUE

THEN

DEST ← temp;

FI;

ELSE

IF (OperandSize = 32 and IA-32e mode active)

THEN

DEST[63:32] ← 0;

FI;

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

CMP—Compare Two Operands

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	I	Valid	Valid	Compare <i>imm8</i> with AL.
3D <i>iw</i>	CMP AX, <i>imm16</i>	I	Valid	Valid	Compare <i>imm16</i> with AX.
3D <i>id</i>	CMP EAX, <i>imm32</i>	I	Valid	Valid	Compare <i>imm32</i> with EAX.
REX.W + 3D <i>id</i>	CMP RAX, <i>imm32</i>	I	Valid	N.E.	Compare <i>imm32</i> sign-extended to 64-bits with RAX.
80 /7 <i>ib</i>	CMP <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m8</i> .
REX + 80 /7 <i>ib</i>	CMP <i>r/m8</i> [*] , <i>imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with <i>r/m8</i> .
81 /7 <i>iw</i>	CMP <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Compare <i>imm16</i> with <i>r/m16</i> .
81 /7 <i>id</i>	CMP <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Compare <i>imm32</i> with <i>r/m32</i> .
REX.W + 81 /7 <i>id</i>	CMP <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Compare <i>imm32</i> sign-extended to 64-bits with <i>r/m64</i> .
83 /7 <i>ib</i>	CMP <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m16</i> .
83 /7 <i>ib</i>	CMP <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m32</i> .
REX.W + 83 /7 <i>ib</i>	CMP <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with <i>r/m64</i> .
38 /r	CMP <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Compare <i>r8</i> with <i>r/m8</i> .
REX + 38 /r	CMP <i>r/m8</i> [*] , <i>r8</i> [*]	MR	Valid	N.E.	Compare <i>r8</i> with <i>r/m8</i> .
39 /r	CMP <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Compare <i>r16</i> with <i>r/m16</i> .
39 /r	CMP <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Compare <i>r32</i> with <i>r/m32</i> .
REX.W + 39 /r	CMP <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Compare <i>r64</i> with <i>r/m64</i> .
3A /r	CMP <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Compare <i>r/m8</i> with <i>r8</i> .
REX + 3A /r	CMP <i>r8</i> [*] , <i>r/m8</i> [*]	RM	Valid	N.E.	Compare <i>r/m8</i> with <i>r8</i> .
3B /r	CMP <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Compare <i>r/m16</i> with <i>r16</i> .
3B /r	CMP <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Compare <i>r/m32</i> with <i>r32</i> .
REX.W + 3B /r	CMP <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Compare <i>r/m64</i> with <i>r64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX (r)	<i>imm8</i>	NA	NA

Description

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

The condition codes used by the *Jcc*, *CMOVcc*, and *SETcc* instructions are based on the results of a CMP instruction. Appendix B, “EFLAGS Condition Codes,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

temp \leftarrow SRC1 – SignExtend(SRC2);

ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
66 OF C2 /r ib CMPPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Compare packed double-precision floating-point values in <i>xmm2/m128</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.128.66.OF.WIG C2 /r ib VCMPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Compare packed double-precision floating-point values in <i>xmm3/m128</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.
VEX.NDS.256.66.OF.WIG C2 /r ib VCMPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Compare packed double-precision floating-point values in <i>ymm3/m256</i> and <i>ymm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand.

Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions

Predic- ate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
EQ	000B	Equal	$A = B$		False	No
LT	001B	Less-than	$A < B$		False	Yes
LE	010B	Less-than-or-equal	$A \leq B$		False	Yes
		Greater than	$A > B$	Swap Operands, Use LT	False	Yes
		Greater-than-or-equal	$A \geq B$	Swap Operands, Use LE	False	Yes
UNORD	011B	Unordered	$A, B = \text{Unordered}$		True	No
NEQ	100B	Not-equal	$A \neq B$		True	No
NLT	101B	Not-less-than	$\text{NOT}(A < B)$		True	Yes

Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

Predicate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
NLE	110B	Not-less-than-or-equal	$\text{NOT}(A \leq B)$		True	Yes
		Not-greater-than	$\text{NOT}(A > B)$	Swap Operands, Use NLT	True	Yes
		Not-greater-than-or-equal	$\text{NOT}(A \geq B)$	Swap Operands, Use NLE	True	Yes
ORD	111B	Ordered	$A, B = \text{Ordered}$		False	No

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that the processors with "CPUID.1H:ECX.AVX = 0" do not implement the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-8. Pseudo-Op and CMPPD Implementation

Pseudo-Op	CMPPD Implementation
CMPEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 0</i>
CMPLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 1</i>
CMPLDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 2</i>
CMPUNORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 3</i>
CMPNEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 4</i>
CMPNLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 5</i>
CMPNLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 6</i>
CMPODPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement, require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Enhanced Comparison Predicate for VEX-Encoded VCMPPD

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Table 3-9. Comparison Predicate for VCMPPD and VCMPPS Instructions

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNaN
			A > B	A < B	A = B	Unordered ¹	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, non-signaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes

Table 3-9. Comparison Predicate for VCMPPD and VCMPPS Instructions (Contd.)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A > B	A < B	A = B	Unordered ¹	
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, non-signaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

NOTES:

1. If either operand A or B is a NAN.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-10, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-10. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMPEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0</i>
VCMPPLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1</i>
VCMPLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 3</i>
VCMPNEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 4</i>
VCMPNLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 5</i>
VCMPNLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 6</i>
VCMPORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 8</i>
VCMPNGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 9</i>
VCMPNGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0CH</i>
VCMPGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0DH</i>
VCMPGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 12H</i>

Table 3-10. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMOUNORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 13H</i>
VCMONEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 14H</i>
VCMONLT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 15H</i>
VCMONLE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 16H</i>
VCMOPORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1AH</i>
VCMFALSE_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1BH</i>
VCMONEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1CH</i>
VCMUGE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1DH</i>
VCMUGT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1EH</i>
VCMPTTRUE_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1FH</i>

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;
- 21: OP5 \leftarrow NLT_UQ;
- 22: OP5 \leftarrow NLE_UQ;
- 23: OP5 \leftarrow ORD_S;
- 24: OP5 \leftarrow EQ_US;
- 25: OP5 \leftarrow NGE_UQ;
- 26: OP5 \leftarrow NGT_UQ;
- 27: OP5 \leftarrow FALSE_OS;
- 28: OP5 \leftarrow NEQ_OS;
- 29: OP5 \leftarrow GE_OQ;

30: OP5 \leftarrow GT_OQ;
 31: OP5 \leftarrow TRUE_US;
 DEFAULT: Reserved;

CMPPD (128-bit Legacy SSE version)

CMP0 \leftarrow SRC1[63:0] OP3 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP3 SRC2[127:64];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 0000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 0000000000000000H; FI;
 DEST[VLMAX-1:128] (Unmodified)

VCMPD (VEX.128 encoded version)

CMP0 \leftarrow SRC1[63:0] OP5 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP5 SRC2[127:64];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 0000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 0000000000000000H; FI;
 DEST[VLMAX-1:128] \leftarrow 0

VCMPD (VEX.256 encoded version)

CMP0 \leftarrow SRC1[63:0] OP5 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP5 SRC2[127:64];
 CMP2 \leftarrow SRC1[191:128] OP5 SRC2[191:128];
 CMP3 \leftarrow SRC1[255:192] OP5 SRC2[255:192];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 0000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 0000000000000000H; FI;
 IF CMP2 = TRUE
 THEN DEST[191:128] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[191:128] \leftarrow 0000000000000000H; FI;
 IF CMP3 = TRUE
 THEN DEST[255:192] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[255:192] \leftarrow 0000000000000000H; FI;

Intel C/C++ Compiler Intrinsic Equivalents

CMPPD for equality: `__m128d _mm_cmpeq_pd(__m128d a, __m128d b)`
 CMPPD for less-than: `__m128d _mm_cmplt_pd(__m128d a, __m128d b)`
 CMPPD for less-than-or-equal: `__m128d _mm_cmple_pd(__m128d a, __m128d b)`
 CMPPD for greater-than: `__m128d _mm_cmpgt_pd(__m128d a, __m128d b)`
 CMPPD for greater-than-or-equal: `__m128d _mm_cmpge_pd(__m128d a, __m128d b)`
 CMPPD for inequality: `__m128d _mm_cmpneq_pd(__m128d a, __m128d b)`
 CMPPD for not-less-than: `__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)`

CMPPD for not-greater-than: `__m128d _mm_cmpngt_pd(__m128d a, __m128d b)`

CMPPD for not-greater-than-or-equal: `__m128d _mm_cmpnge_pd(__m128d a, __m128d b)`

CMPPD for ordered: `__m128d _mm_cmpord_pd(__m128d a, __m128d b)`

CMPPD for unordered: `__m128d _mm_cmpunord_pd(__m128d a, __m128d b)`

CMPPD for not-less-than-or-equal: `__m128d _mm_cmpnle_pd(__m128d a, __m128d b)`

VCMPD: `__m256 _mm256_cmp_pd(__m256 a, __m256 b, const int imm)`

VCMPD: `__m128 _mm_cmp_pd(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 2.

CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
OF C2 /r ib CMPPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Compare packed single-precision floating-point values in <i>xmm2/mem</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.128.OF.WIG C2 /r ib VCMPPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Compare packed single-precision floating-point values in <i>xmm3/m128</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.
VEX.NDS.256.OF.WIG C2 /r ib VCMPPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Compare packed single-precision floating-point values in <i>ymm3/m256</i> and <i>ymm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of $+0.0$ and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX = 0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with “CPUID.1H:ECX.AVX = 0”. See Table 3-11. Compiler should treat reserved Imm8 values as illegal syntax.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Table 3-11. Pseudo-Ops and CMPPS

Pseudo-Op	Implementation
CMPEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 0</i>
CMPLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 1</i>
CMPLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 2</i>
CMPUNORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 3</i>
CMPNEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 4</i>
CMPNLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 5</i>
CMPNLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 6</i>
CMPORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented by processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Enhanced Comparison Predicate for VEX-Encoded VCMPPS

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-12, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-12. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 2</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 3</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 4</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 5</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 6</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 8</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 9</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0AH</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0BH</i>

Table 3-12. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQ_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0CH</i>
VCMERGEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0DH</i>
VCMPGTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0EH</i>
VCMPTTRUEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 11H</i>
VCMPLT_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 12H</i>
VCMPEQ_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 13H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 14H</i>
VCMPLT_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 15H</i>
VCMPLT_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 16H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 18H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 19H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1AH</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1BH</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1CH</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1DH</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1EH</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1FH</i>

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;

21: OP5 \leftarrow NLT_UQ;
 22: OP5 \leftarrow NLE_UQ;
 23: OP5 \leftarrow ORD_S;
 24: OP5 \leftarrow EQ_US;
 25: OP5 \leftarrow NGE_UQ;
 26: OP5 \leftarrow NGT_UQ;
 27: OP5 \leftarrow FALSE_OS;
 28: OP5 \leftarrow NEQ_OS;
 29: OP5 \leftarrow GE_OQ;
 30: OP5 \leftarrow GT_OQ;
 31: OP5 \leftarrow TRUE_US;
 DEFAULT: Reserved

EASC;

CMPPS (128-bit Legacy SSE version)

CMP0 \leftarrow SRC1[31:0] OP3 SRC2[31:0];
 CMP1 \leftarrow SRC1[63:32] OP3 SRC2[63:32];
 CMP2 \leftarrow SRC1[95:64] OP3 SRC2[95:64];
 CMP3 \leftarrow SRC1[127:96] OP3 SRC2[127:96];
 IF CMP0 = TRUE
 THEN DEST[31:0] \leftarrow FFFFFFFFH;
 ELSE DEST[31:0] \leftarrow 00000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[63:32] \leftarrow FFFFFFFFH;
 ELSE DEST[63:32] \leftarrow 00000000H; FI;
 IF CMP2 = TRUE
 THEN DEST[95:64] \leftarrow FFFFFFFFH;
 ELSE DEST[95:64] \leftarrow 00000000H; FI;
 IF CMP3 = TRUE
 THEN DEST[127:96] \leftarrow FFFFFFFFH;
 ELSE DEST[127:96] \leftarrow 00000000H; FI;
 DEST[VLMAX-1:128] (Unmodified)

VCMPSS (VEX.128 encoded version)

CMP0 \leftarrow SRC1[31:0] OP5 SRC2[31:0];
 CMP1 \leftarrow SRC1[63:32] OP5 SRC2[63:32];
 CMP2 \leftarrow SRC1[95:64] OP5 SRC2[95:64];
 CMP3 \leftarrow SRC1[127:96] OP5 SRC2[127:96];
 IF CMP0 = TRUE
 THEN DEST[31:0] \leftarrow FFFFFFFFH;
 ELSE DEST[31:0] \leftarrow 00000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[63:32] \leftarrow FFFFFFFFH;
 ELSE DEST[63:32] \leftarrow 00000000H; FI;
 IF CMP2 = TRUE
 THEN DEST[95:64] \leftarrow FFFFFFFFH;
 ELSE DEST[95:64] \leftarrow 00000000H; FI;
 IF CMP3 = TRUE
 THEN DEST[127:96] \leftarrow FFFFFFFFH;
 ELSE DEST[127:96] \leftarrow 00000000H; FI;
 DEST[VLMAX-1:128] \leftarrow 0

VCMPPS (VEX.256 encoded version)

CMP0 \leftarrow SRC1[31:0] OP5 SRC2[31:0];
 CMP1 \leftarrow SRC1[63:32] OP5 SRC2[63:32];
 CMP2 \leftarrow SRC1[95:64] OP5 SRC2[95:64];
 CMP3 \leftarrow SRC1[127:96] OP5 SRC2[127:96];
 CMP4 \leftarrow SRC1[159:128] OP5 SRC2[159:128];
 CMP5 \leftarrow SRC1[191:160] OP5 SRC2[191:160];
 CMP6 \leftarrow SRC1[223:192] OP5 SRC2[223:192];
 CMP7 \leftarrow SRC1[255:224] OP5 SRC2[255:224];
 IF CMP0 = TRUE
 THEN DEST[31:0] \leftarrow FFFFFFFFH;
 ELSE DEST[31:0] \leftarrow 00000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[63:32] \leftarrow FFFFFFFFH;
 ELSE DEST[63:32] \leftarrow 00000000H; FI;
 IF CMP2 = TRUE
 THEN DEST[95:64] \leftarrow FFFFFFFFH;
 ELSE DEST[95:64] \leftarrow 00000000H; FI;
 IF CMP3 = TRUE
 THEN DEST[127:96] \leftarrow FFFFFFFFH;
 ELSE DEST[127:96] \leftarrow 00000000H; FI;
 IF CMP4 = TRUE
 THEN DEST[159:128] \leftarrow FFFFFFFFH;
 ELSE DEST[159:128] \leftarrow 00000000H; FI;
 IF CMP5 = TRUE
 THEN DEST[191:160] \leftarrow FFFFFFFFH;
 ELSE DEST[191:160] \leftarrow 00000000H; FI;
 IF CMP6 = TRUE
 THEN DEST[223:192] \leftarrow FFFFFFFFH;
 ELSE DEST[223:192] \leftarrow 00000000H; FI;
 IF CMP7 = TRUE
 THEN DEST[255:224] \leftarrow FFFFFFFFH;
 ELSE DEST[255:224] \leftarrow 00000000H; FI;

Intel C/C++ Compiler Intrinsic Equivalents

CMPPS for equality: `__m128_mm_cmpeq_ps(__m128 a, __m128 b)`
 CMPPS for less-than: `__m128_mm_cmplt_ps(__m128 a, __m128 b)`
 CMPPS for less-than-or-equal: `__m128_mm_cmple_ps(__m128 a, __m128 b)`
 CMPPS for greater-than: `__m128_mm_cmpgt_ps(__m128 a, __m128 b)`
 CMPPS for greater-than-or-equal: `__m128_mm_cmpge_ps(__m128 a, __m128 b)`
 CMPPS for inequality: `__m128_mm_cmpneq_ps(__m128 a, __m128 b)`
 CMPPS for not-less-than: `__m128_mm_cmpnlt_ps(__m128 a, __m128 b)`
 CMPPS for not-greater-than: `__m128_mm_cmpngt_ps(__m128 a, __m128 b)`
 CMPPS for not-greater-than-or-equal: `__m128_mm_cmpnge_ps(__m128 a, __m128 b)`
 CMPPS for ordered: `__m128_mm_cmpord_ps(__m128 a, __m128 b)`
 CMPPS for unordered: `__m128_mm_cmpunord_ps(__m128 a, __m128 b)`
 CMPPS for not-less-than-or-equal: `__m128_mm_cmpnle_ps(__m128 a, __m128 b)`
 VCMPPS: `__m256_mm256_cmp_ps(__m256 a, __m256 b, const int imm)`
 VCMPPS: `__m128_mm_cmp_ps(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 2.

CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A6	CMPS <i>m8, m8</i>	NP	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R)ESI to byte at address (R)EDI. The status flags are set accordingly.
A7	CMPS <i>m16, m16</i>	NP	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R)ESI with word at address (R)EDI. The status flags are set accordingly.
A7	CMPS <i>m32, m32</i>	NP	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R)ESI at dword at address (R)EDI. The status flags are set accordingly.
REX.W + A7	CMPS <i>m64, m64</i>	NP	Valid	N.E.	Compares quadword at address (R)ESI with quadword at address (R)EDI and sets the status flags accordingly.
A6	CMPSB	NP	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R)ESI with byte at address (R)EDI. The status flags are set accordingly.
A7	CMPSW	NP	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R)ESI with word at address (R)EDI. The status flags are set accordingly.
A7	CMPSD	NP	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R)ESI with dword at address (R)EDI. The status flags are set accordingly.
REX.W + A7	CMPSQ	NP	Valid	N.E.	Compares quadword at address (R)ESI with quadword at address (R)EDI and sets the status flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the

instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

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

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
temp ← SRC1 - SRC2;
SetStatusFlags(temp);
```

```
IF (64-Bit Mode)
  THEN
    IF (Byte comparison)
      THEN IF DF = 0
        THEN
          (R)ESI ← (R)ESI + 1;
          (R)EDI ← (R)EDI + 1;
        ELSE
          (R)ESI ← (R)ESI - 1;
          (R)EDI ← (R)EDI - 1;
        FI;
      ELSE IF (Word comparison)
        THEN IF DF = 0
          THEN
            (R)ESI ← (R)ESI + 2;
            (R)EDI ← (R)EDI + 2;
          ELSE
            (R)ESI ← (R)ESI - 2;
            (R)EDI ← (R)EDI - 2;
          FI;
        ELSE IF (Doubleword comparison)
```

```

    THEN IF DF = 0
        THEN
            (R)E)SI  $\leftarrow$  (R)E)SI + 4;
            (R)E)DI  $\leftarrow$  (R)E)DI + 4;
        ELSE
            (R)E)SI  $\leftarrow$  (R)E)SI - 4;
            (R)E)DI  $\leftarrow$  (R)E)DI - 4;
        FI;
    ELSE (* Quadword comparison *)
        THEN IF DF = 0
            (R)E)SI  $\leftarrow$  (R)E)SI + 8;
            (R)E)DI  $\leftarrow$  (R)E)DI + 8;
        ELSE
            (R)E)SI  $\leftarrow$  (R)E)SI - 8;
            (R)E)DI  $\leftarrow$  (R)E)DI - 8;
        FI;
    FI;
ELSE (* Non-64-bit Mode *)
    IF (byte comparison)
        THEN IF DF = 0
            THEN
                (E)SI  $\leftarrow$  (E)SI + 1;
                (E)DI  $\leftarrow$  (E)DI + 1;
            ELSE
                (E)SI  $\leftarrow$  (E)SI - 1;
                (E)DI  $\leftarrow$  (E)DI - 1;
            FI;
        ELSE IF (word comparison)
            THEN IF DF = 0
                (E)SI  $\leftarrow$  (E)SI + 2;
                (E)DI  $\leftarrow$  (E)DI + 2;
            ELSE
                (E)SI  $\leftarrow$  (E)SI - 2;
                (E)DI  $\leftarrow$  (E)DI - 2;
            FI;
        ELSE (* Doubleword comparison *)
            THEN IF DF = 0
                (E)SI  $\leftarrow$  (E)SI + 4;
                (E)DI  $\leftarrow$  (E)DI + 4;
            ELSE
                (E)SI  $\leftarrow$  (E)SI - 4;
                (E)DI  $\leftarrow$  (E)DI - 4;
            FI;
        FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

- #GP(0) 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

CMPSD—Compare Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD <i>xmm1</i> , <i>xmm2/m64</i> , <i>imm8</i>	RMI	V/V	SSE2	Compare low double-precision floating-point value in <i>xmm2/m64</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m64</i> , <i>imm8</i>	RVMI	V/V	AVX	Compare low double precision floating-point value in <i>xmm3/m64</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of $+0.0$ and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX = 0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with “CPUID.1H:ECX.AVX = 0”. See Table 3-13. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-13. Pseudo-Ops and CMPSD

Pseudo-Op	Implementation
CMPEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 0</i>
CMPLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 1</i>
CMPLESD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 2</i>
CMPUNORDSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 3</i>
CMPNEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 4</i>
CMPNLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 5</i>
CMPNLESD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 6</i>
CMPORDSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented in the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Enhanced Comparison Predicate for VEX-Encoded VCMPSD

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-14, where the notations of reg1, reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-14. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPSEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPNEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMPNLTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMPNLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPSEQ_QSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPNGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPNGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_QSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0CH</i>
VCMPGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0DH</i>
VCMPGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0EH</i>

Table 3-14. Pseudo-Op and VCMPSD Implementation (Contd.)

Pseudo-Op	CMPSD Implementation
VCMPTRUESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0FH</i>
VCMPSEQ_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 10H</i>
VCMPLE_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 12H</i>
VCMPUNORD_SSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 17H</i>
VCMPSEQ_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1EH</i>
VCMPTTRUE_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1FH</i>

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;
- 21: OP5 \leftarrow NLT_UQ;
- 22: OP5 \leftarrow NLE_UQ;
- 23: OP5 \leftarrow ORD_S;
- 24: OP5 \leftarrow EQ_US;

25: OP5 \leftarrow NGE_UQ;
 26: OP5 \leftarrow NGT_UQ;
 27: OP5 \leftarrow FALSE_OS;
 28: OP5 \leftarrow NEQ_OS;
 29: OP5 \leftarrow GE_OQ;
 30: OP5 \leftarrow GT_OQ;
 31: OP5 \leftarrow TRUE_US;
 DEFAULT: Reserved
 ESAC;

CMPSD (128-bit Legacy SSE version)

CMP0 \leftarrow DEST[63:0] OP3 SRC[63:0];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFF FFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 00000000 00000000H; FI;
 DEST[VLMAX-1:64] (Unmodified)

VCMPSD (VEX.128 encoded version)

CMP0 \leftarrow SRC1[63:0] OP5 SRC2[63:0];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFF FFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 00000000 00000000H; FI;
 DEST[127:64] \leftarrow SRC1[127:64]
 DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalents

CMPSD for equality: `__m128d_mm_cmpeq_sd(__m128d a, __m128d b)`
 CMPSD for less-than: `__m128d_mm_cmplt_sd(__m128d a, __m128d b)`
 CMPSD for less-than-or-equal: `__m128d_mm_cmple_sd(__m128d a, __m128d b)`
 CMPSD for greater-than: `__m128d_mm_cmpgt_sd(__m128d a, __m128d b)`
 CMPSD for greater-than-or-equal: `__m128d_mm_cmpge_sd(__m128d a, __m128d b)`
 CMPSD for inequality: `__m128d_mm_cmpneq_sd(__m128d a, __m128d b)`
 CMPSD for not-less-than: `__m128d_mm_cmpnlt_sd(__m128d a, __m128d b)`
 CMPSD for not-greater-than: `__m128d_mm_cmpngt_sd(__m128d a, __m128d b)`
 CMPSD for not-greater-than-or-equal: `__m128d_mm_cmpnge_sd(__m128d a, __m128d b)`
 CMPSD for ordered: `__m128d_mm_cmpord_sd(__m128d a, __m128d b)`
 CMPSD for unordered: `__m128d_mm_cmpunord_sd(__m128d a, __m128d b)`
 CMPSD for not-less-than-or-equal: `__m128d_mm_cmpnle_sd(__m128d a, __m128d b)`
 VCMPSD: `__m128_mm_cmp_sd(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 3.

CMPSS—Compare Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS <i>xmm1</i> , <i>xmm2/m32</i> , <i>imm8</i>	RMI	V/V	SSE	Compare low single-precision floating-point value in <i>xmm2/m32</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i> , <i>imm8</i>	RVMI	V/V	AVX	Compare low single precision floating-point value in <i>xmm3/m32</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a double-word mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, since a mask of all 0s corresponds to a floating-point value of $+0.0$ and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX = 0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with “CPUID.1H:ECX.AVX = 0”. See Table 3-15. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-15. Pseudo-Ops and CMPSS

Pseudo-Op	CMPSS Implementation
CMPEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 0</i>
CMPLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 1</i>
CMPLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 2</i>
CMPUNORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 3</i>
CMPNEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 4</i>
CMPNLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 5</i>
CMPNLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 6</i>
CMPORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented in the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Enhanced Comparison Predicate for VEX-Encoded VCMPSD

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-16, where the notations of reg1, reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-16. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPLSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPLUNORDSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPLNEQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMPLNLTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMPLNLESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPLORDSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPLNGESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPLNGTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0AH</i>
VCMPLFALSESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0BH</i>
VCMPLNEQ_OQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0CH</i>
VCMPLGESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0DH</i>
VCMPLGTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0EH</i>

Table 3-16. Pseudo-Op and VCMPS Implementation (Contd.)

Pseudo-Op	CMPS Implementation
VCMPTTRUESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0FH</i>
VCMPSEQ_OSSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 10H</i>
VCMPLE_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 12H</i>
VCMPUNORD_SSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 17H</i>
VCMPSEQ_USSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1EH</i>
VCMPTTRUE_USSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1FH</i>

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;
- 21: OP5 \leftarrow NLT_UQ;
- 22: OP5 \leftarrow NLE_UQ;
- 23: OP5 \leftarrow ORD_S;
- 24: OP5 \leftarrow EQ_US;

25: OP5 \leftarrow NGE_UQ;
 26: OP5 \leftarrow NGT_UQ;
 27: OP5 \leftarrow FALSE_OS;
 28: OP5 \leftarrow NEQ_OS;
 29: OP5 \leftarrow GE_OQ;
 30: OP5 \leftarrow GT_OQ;
 31: OP5 \leftarrow TRUE_US;
 DEFAULT: Reserved

ESAC;

CMPSS (128-bit Legacy SSE version)

CMP0 \leftarrow DEST[31:0] OP3 SRC[31:0];
 IF CMP0 = TRUE
 THEN DEST[31:0] \leftarrow FFFFFFFFH;
 ELSE DEST[31:0] \leftarrow 00000000H; FI;
 DEST[VLMAX-1:32] (Unmodified)

VCMPSS (VEX.128 encoded version)

CMP0 \leftarrow SRC1[31:0] OP5 SRC2[31:0];
 IF CMP0 = TRUE
 THEN DEST[31:0] \leftarrow FFFFFFFFH;
 ELSE DEST[31:0] \leftarrow 00000000H; FI;
 DEST[127:32] \leftarrow SRC1[127:32]
 DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalents

CMPSS for equality: `__m128 _mm_cmpeq_ss(__m128 a, __m128 b)`
 CMPSS for less-than: `__m128 _mm_cmlt_ss(__m128 a, __m128 b)`
 CMPSS for less-than-or-equal: `__m128 _mm_cmple_ss(__m128 a, __m128 b)`
 CMPSS for greater-than: `__m128 _mm_cmpgt_ss(__m128 a, __m128 b)`
 CMPSS for greater-than-or-equal: `__m128 _mm_cmpge_ss(__m128 a, __m128 b)`
 CMPSS for inequality: `__m128 _mm_cmpneq_ss(__m128 a, __m128 b)`
 CMPSS for not-less-than: `__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)`
 CMPSS for not-greater-than: `__m128 _mm_cmpngt_ss(__m128 a, __m128 b)`
 CMPSS for not-greater-than-or-equal: `__m128 _mm_cmpnge_ss(__m128 a, __m128 b)`
 CMPSS for ordered: `__m128 _mm_cmpord_ss(__m128 a, __m128 b)`
 CMPSS for unordered: `__m128 _mm_cpunord_ss(__m128 a, __m128 b)`
 CMPSS for not-less-than-or-equal: `__m128 _mm_cmpnle_ss(__m128 a, __m128 b)`
 VCMPPSS: `__m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 3.

CMPXCHG—Compare and Exchange

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF B0/ <i>r</i> CMPXCHG <i>r/m8, r8</i>	MR	Valid	Valid*	Compare AL with <i>r/m8</i> . If equal, ZF is set and <i>r8</i> is loaded into <i>r/m8</i> . Else, clear ZF and load <i>r/m8</i> into AL.
REX + OF B0/ <i>r</i> CMPXCHG <i>r/m8**, r8</i>	MR	Valid	N.E.	Compare AL with <i>r/m8</i> . If equal, ZF is set and <i>r8</i> is loaded into <i>r/m8</i> . Else, clear ZF and load <i>r/m8</i> into AL.
OF B1/ <i>r</i> CMPXCHG <i>r/m16, r16</i>	MR	Valid	Valid*	Compare AX with <i>r/m16</i> . If equal, ZF is set and <i>r16</i> is loaded into <i>r/m16</i> . Else, clear ZF and load <i>r/m16</i> into AX.
OF B1/ <i>r</i> CMPXCHG <i>r/m32, r32</i>	MR	Valid	Valid*	Compare EAX with <i>r/m32</i> . If equal, ZF is set and <i>r32</i> is loaded into <i>r/m32</i> . Else, clear ZF and load <i>r/m32</i> into EAX.
REX.W + OF B1/ <i>r</i> CMPXCHG <i>r/m64, r64</i>	MR	Valid	N.E.	Compare RAX with <i>r/m64</i> . If equal, ZF is set and <i>r64</i> is loaded into <i>r/m64</i> . Else, clear ZF and load <i>r/m64</i> into RAX.

NOTES:

* See the IA-32 Architecture Compatibility section below.

** In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (<i>r, w</i>)	ModRM:reg (<i>r</i>)	NA	NA

Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

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

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed *)

TEMP ← DEST

IF accumulator = TEMP

THEN

ZF ← 1;

DEST ← SRC;

ELSE

```

ZF ← 0;
accumulator ← TEMP;
DEST ← TEMP;

```

```

FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C7 /1 <i>m64</i> CMPXCHG8B <i>m64</i>	M	Valid	Valid*	Compare EDX:EAX with <i>m64</i> . If equal, set ZF and load ECX:EBX into <i>m64</i> . Else, clear ZF and load <i>m64</i> into EDX:EAX.
REX.W + OF C7 /1 <i>m128</i> CMPXCHG16B <i>m128</i>	M	Valid	N.E.	Compare RDX:RAX with <i>m128</i> . If equal, set ZF and load RCX:RBX into <i>m128</i> . Else, clear ZF and load <i>m128</i> into RDX:RAX.

NOTES:

*See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64bits of a 128-bit value.

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

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-175.

IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

Operation

```

IF (64-Bit Mode and OperandSize = 64)
  THEN
    TEMP128 ← DEST
    IF (RDX:RAX = TEMP128)
      THEN
        ZF ← 1;
        DEST ← RCX:RBX;
      ELSE
        ZF ← 0;
        RDX:RAX ← TEMP128;
        DEST ← TEMP128;
      FI;
    FI;

```

```

ELSE
    TEMP64 ← DEST;
    IF (EDX:EAX = TEMP64)
        THEN
            ZF ← 1;
            DEST ← ECX:EBX;
        ELSE
            ZF ← 0;
            EDX:EAX ← TEMP64;
            DEST ← TEMP64;
        FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#UD	If the destination is not a memory operand.
#GP(0)	If the destination is located in a non-writable 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

#UD	If the destination operand is not a memory location.
#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.

Virtual-8086 Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary. If CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0.
#UD	If the destination operand is not a memory location.

#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.

COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 2F /r COMISD <i>xmm1</i> , <i>xmm2/mem64</i>	RM	V/V	SSE2	Compare low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.
VEX.LIG.66.OF.WIG 2F /r VCOMISD <i>xmm1</i> , <i>xmm2/mem64</i>	RM	V/V	AVX	Compare low double precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```

RESULT ← OrderedCompare(DEST[63:0] ≠ SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED:      ZF,PF,CF ← 111;
  GREATER_THAN:   ZF,PF,CF ← 000;
  LESS_THAN:      ZF,PF,CF ← 001;
  EQUAL:          ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }

```

Intel C/C++ Compiler Intrinsic Equivalents

```

int __mm_comieq_sd (__m128d a, __m128d b)
int __mm_comilt_sd (__m128d a, __m128d b)
int __mm_comile_sd (__m128d a, __m128d b)
int __mm_comigt_sd (__m128d a, __m128d b)
int __mm_comige_sd (__m128d a, __m128d b)
int __mm_comineq_sd (__m128d a, __m128d b)

```

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv \neq 1111B.

COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 2F /r COMISS <i>xmm1</i> , <i>xmm2/mem32</i>	RM	V/V	SSE	Compare low single-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2F /r VCOMISS <i>xmm1</i> , <i>xmm2/mem32</i>	RM	V/V	AVX	Compare low single precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

Operand 1 is an XMM register; Operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```

RESULT ← OrderedCompare(SRC1[31:0] ≠ SRC2[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED:      ZF,PF,CF ← 111;
  GREATER_THAN:   ZF,PF,CF ← 000;
  LESS_THAN:      ZF,PF,CF ← 001;
  EQUAL:          ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0; }

```

Intel C/C++ Compiler Intrinsic Equivalents

```

int _mm_comieq_ss (__m128 a, __m128 b)
int _mm_comilt_ss (__m128 a, __m128 b)
int _mm_comile_ss (__m128 a, __m128 b)
int _mm_comigt_ss (__m128 a, __m128 b)
int _mm_comige_ss (__m128 a, __m128 b)
int _mm_comineq_ss (__m128 a, __m128 b)

```

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv \neq 1111B.

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF A2	CPUID	NP	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.¹ The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-17 shows information returned, depending on the initial value loaded into the EAX register. Table 3-18 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
CPUID.EAX = 0CH (* INVALID: Returns the same information as CPUID.EAX = 0BH. *)
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 07H (*Returns EAX=EBX=ECX=EDX=0. *)
```

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Table 3-17. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
Basic CPUID Information		
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 3-18) "Genu" "ntel" "inel"
01H	EAX EBX ECX EDX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5) Bits 07-00: Brand Index Bits 15-08: CLFLUSH line size (Value * 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID Feature Information (see Figure 3-6 and Table 3-19) Feature Information (see Figure 3-7 and Table 3-20) NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-21) Cache and TLB Information Cache and TLB Information Cache and TLB Information
03H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature. CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLE.BOOT_NT4[bit 22] = 0 (default).
Deterministic Cache Parameters Leaf		
04H	EAX	NOTES: Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 3-183." Bits 04-00: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved Bits 07-05: Cache Level (starts at 1) Bit 08: Self Initializing cache level (does not need SW initialization) Bit 09: Fully Associative cache

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		<p>Bits 13-10: Reserved</p> <p>Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache**, ***</p> <p>Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****</p> <p>EBX Bits 11-00: L = System Coherency Line Size**</p> <p>Bits 21-12: P = Physical Line partitions**</p> <p>Bits 31-22: W = Ways of associativity**</p> <p>ECX Bits 31-00: S = Number of Sets**</p> <p>EDX Bit 0: Write-Back Invalidate/Invalidate</p> <p>0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache.</p> <p>1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.</p> <p>Bit 1: Cache Inclusiveness</p> <p>0 = Cache is not inclusive of lower cache levels.</p> <p>1 = Cache is inclusive of lower cache levels.</p> <p>Bit 2: Complex Cache Indexing</p> <p>0 = Direct mapped cache.</p> <p>1 = A complex function is used to index the cache, potentially using all address bits.</p> <p>Bits 31-03: Reserved = 0</p> <p>NOTES:</p> <p>* If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0.</p> <p>** Add one to the return value to get the result.</p> <p>***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache</p> <p>**** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.</p> <p>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p>
<i>MONITOR/MWAIT Leaf</i>		
05H	EAX EBX ECX	<p>Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity)</p> <p>Bits 31-16: Reserved = 0</p> <p>Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity)</p> <p>Bits 31-16: Reserved = 0</p> <p>Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported</p> <p>Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled</p> <p>Bits 31 - 02: Reserved</p>

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT Bits 07 - 04: Number of C1* sub C-states supported using MWAIT Bits 11 - 08: Number of C2* sub C-states supported using MWAIT Bits 15 - 12: Number of C3* sub C-states supported using MWAIT Bits 19 - 16: Number of C4* sub C-states supported using MWAIT Bits 23 - 20: Number of C5* sub C-states supported using MWAIT Bits 27 - 24: Number of C6* sub C-states supported using MWAIT Bits 31 - 28: Number of C7* sub C-states supported using MWAIT NOTE: * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
<i>Thermal and Power Management Leaf</i>		
06H	EAX	Bit 00: Digital temperature sensor is supported if set Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bits 31 - 15: Reserved
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H). Bits 31 - 04: Reserved = 0
	EDX	Reserved = 0
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>		
07H	Sub-leaf 0 (Input ECX = 0). *	
	EAX	Bits 31-00: Reports the maximum input value for supported leaf 7 sub-leaves.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: Reserved Bit 03: BMI1 Bit 04: HLE Bit 05: AVX2 Bit 06: Reserved Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2 Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM Bit 12: Supports Platform Quality of Service Monitoring (PQM) capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: Reserved. Bit 15: Supports Platform Quality of Service Enforcement (PQE) capability if 1. Bits 17:16: Reserved Bit 18: RDSEED Bit 19: ADX Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bits 24:21: Reserved Bit 25: Intel Processor Trace Bits 31:26: Reserved
	ECX	Bit 00: PREFETCHWT1 Bits 02:01: Reserved Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions) Bits 31:05: Reserved
	EDX	Reserved
NOTE: * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.		
<i>Direct Cache Access Information Leaf</i>		
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H)
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
<i>Architectural Performance Monitoring Leaf</i>		
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor Bits 23 - 16: Bit width of general-purpose, performance monitoring counter Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: Core cycle event not available if 1 Bit 01: Instruction retired event not available if 1 Bit 02: Reference cycles event not available if 1 Bit 03: Last-level cache reference event not available if 1 Bit 04: Last-level cache misses event not available if 1 Bit 05: Branch instruction retired event not available if 1 Bit 06: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0
	ECX	Reserved = 0
	EDX	Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1) Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1) Reserved = 0
<i>Extended Topology Enumeration Leaf</i>		
OBH		NOTES: Most of Leaf OBH output depends on the initial value in ECX. The EDX output of leaf OBH is always valid and does not vary with input value in ECX. Output value in ECX[7:0] always equals input value in ECX[7:0]. For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0. If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > n also return 0 in ECX[15:8].
	EAX	Bits 04-00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31-05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input Bits 15 - 08: Level type***. Bits 31 - 16:: Reserved.
	EDX	Bits 31- 00: x2APIC ID the current logical processor.
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: invalid 1: SMT 2: Core 3-255: Reserved
<i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>		
0DH		NOTES: Leaf 0DH main leaf (ECX = 0).

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EAX	Bits 31-00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state Bit 01: SSE state Bit 02: AVX state Bits 04 - 03: Reserved Bit 07 - 05: AVX-512 state Bits 31- 08: Reserved
	EBX	Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCR0.
	EDX	Bit 31-00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31- 00: Reserved
<i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>		
0DH	EAX	Bits 31-04: Reserved Bit 00: XSAVEOPT is available Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set Bit 02: Supports XGETBV with ECX = 1 if set Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set
	EBX	Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCR0 IA32_XSS.
	ECX	Bits 31-00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 02-00: used for XCR0 Bits 04 - 03: Reserved Bit 07 - 05: used for XCR0 Bits 31-08: Reserved
	EDX	Bits 31-00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31- 00: Reserved
<i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)</i>		
0DH	NOTES: Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n ($0 \leq n \leq 31$) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n ($32 \leq n \leq 63$) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].	
	EAX	Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n.
	EBX	Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 0 is set if the bit <i>n</i> (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit <i>n</i> is instead supported in XCRO. Bits 31-1 are reserved. This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*.
	EDX	This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*; otherwise it is reserved.
<i>Platform QoS Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)</i>		
0FH		NOTES: Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX
	EAX	Reserved.
	EBX	Bits 31-0: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache QoS Monitoring if 1. Bits 31:02: Reserved
<i>L3 Cache QoS Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)</i>		
0FH		NOTES: Leaf 0FH output depends on the initial value in ECX.
	EAX	Reserved.
	EBX	Bits 31-0: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bits 31:01: Reserved
<i>Platform QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>		
10H		NOTES: Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EDX
	EAX	Reserved.
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache QoS Enforcement if 1. Bits 31:02: Reserved
	ECX	Reserved.
	EDX	Reserved.
<i>L3 Cache QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 1)</i>		
10H		NOTES: Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 4:0: Length of the capacity bit mask for the corresponding ResID. Bits 31:05: Reserved
	EBX	Bits 31-0: Bit-granular map of isolation/contention of allocation units.
	ECX	Bit 00: Reserved. Bit 01: Updates of COS should be infrequent if 1. Bits 31:02: Reserved

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 15:0: Highest COS number supported for this ResID. Bits 31:16: Reserved
<i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>		
14H		<p>NOTES: Leaf 14H main leaf (ECX = 0).</p> <p>EAX Bits 31-0: Reports the maximum number sub-leaves that are supported in leaf 14H.</p> <p>EBX Bit 00: If 1, Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bits 31- 01: Reserved</p> <p>ECX Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOffsetTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 30:02: Reserved Bit 31: If 1, Generated packets which contain IP payloads have LIP values, which include the CS base component.</p> <p>EDX Bits 31- 00: Reserved</p>
<i>Time Stamp Counter/Core Crystal Clock Information-leaf</i>		
15H		<p>NOTES: If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</p> <p>EAX Bits 31:0: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.</p> <p>EBX Bits 31-0: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.</p> <p>ECX Bits 31:0: Reserved = 0.</p> <p>EDX Bits 31:0: Reserved = 0.</p>
<i>Processor Frequency Information Leaf</i>		
16H	EAX	Bits 15:0: Processor Base Frequency (in MHz). Bits 31:16: Reserved = 0
	EBX	Bits 15:0: Maximum Frequency (in MHz). Bits 31:16: Reserved = 0
	ECX	Bits 15:0: Bus (Reference) Frequency (in MHz). Bits 31:16: Reserved = 0
	EDX	Reserved
		<p>NOTES: * Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.</p> <p>While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.</p>
<i>Unimplemented CPUID Leaf Functions</i>		

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
40000000H - 4FFFFFFFH	Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.	
Extended Function CPUID Information		
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 3-18).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode Bits 04-01 Reserved Bit 05: LZCNT Bits 07-06 Reserved Bit 08: PREFETCHW Bits 31-09 Reserved
	EDX	Bits 10-00: Reserved Bit 11: SYSCALL/SYSRET available in 64-bit mode Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 25-21: Reserved = 0 Bit 26: 1-GByte pages are available if 1 Bit 27: RDTSCP and IA32_TSC_AUX are available if 1 Bits 28: Reserved = 0 Bit 29: Intel® 64 Architecture available if 1 Bits 31-30: Reserved = 0
80000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000004H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000005H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
80000006H	EAX	Reserved = 0
	EBX	Reserved = 0

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 07-00: Cache Line size in bytes Bits 11-08: Reserved Bits 15-12: L2 Associativity field *
	EDX	Bits 31-16: Cache size in 1K units Reserved = 0 NOTES: * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative
80000007H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Bits 07-00: Reserved = 0 Bit 08: Invariant TSC available if 1 Bits 31-09: Reserved = 0
80000008H	EAX	Linear/Physical Address size Bits 07-00: #Physical Address Bits* Bits 15-8: #Linear Address Bits Bits 31-16: Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
		NOTES: * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 3-18) and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ← 756e6547h (* "Genu", with G in the low eight bits of BL *)

EDX ← 49656e69h (* "inel", with i in the low eight bits of DL *)

ECX ← 6c65746eh (* "ntel", with n in the low eight bits of CL *)

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-5). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-18 for available processor type values. Stepping IDs are provided as needed.

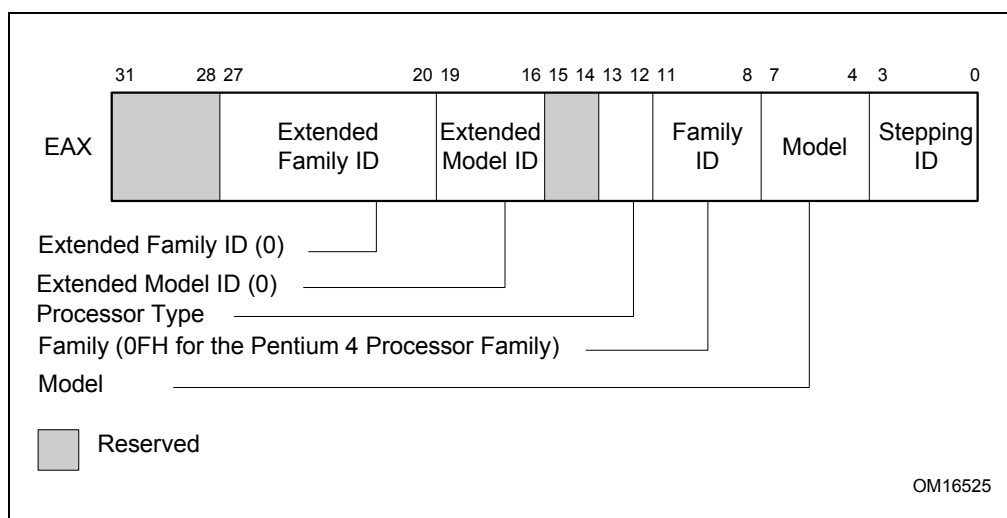


Figure 3-5. Version Information Returned by CPUID in EAX

Table 3-18. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive™ Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

NOTE

See Chapter 17 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
  THEN DisplayFamily = Family_ID;
```

```
ELSE DisplayFamily = Extended_Family_ID + Family_ID;
(* Right justify and zero-extend 4-bit field. *)
```

```
FI;
```

```
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
```

```
THEN DisplayModel = (Extended_Model_ID << 4) + Model_ID;
```

```
(* Right justify and zero-extend 4-bit field; display Model_ID as HEX field. *)
```

```
ELSE DisplayModel = Model_ID;
```

```
FI;
```

```
(* Show DisplayModel as HEX field. *)
```

INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-6 and Table 3-19 show encodings for ECX.
- Figure 3-7 and Table 3-20 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

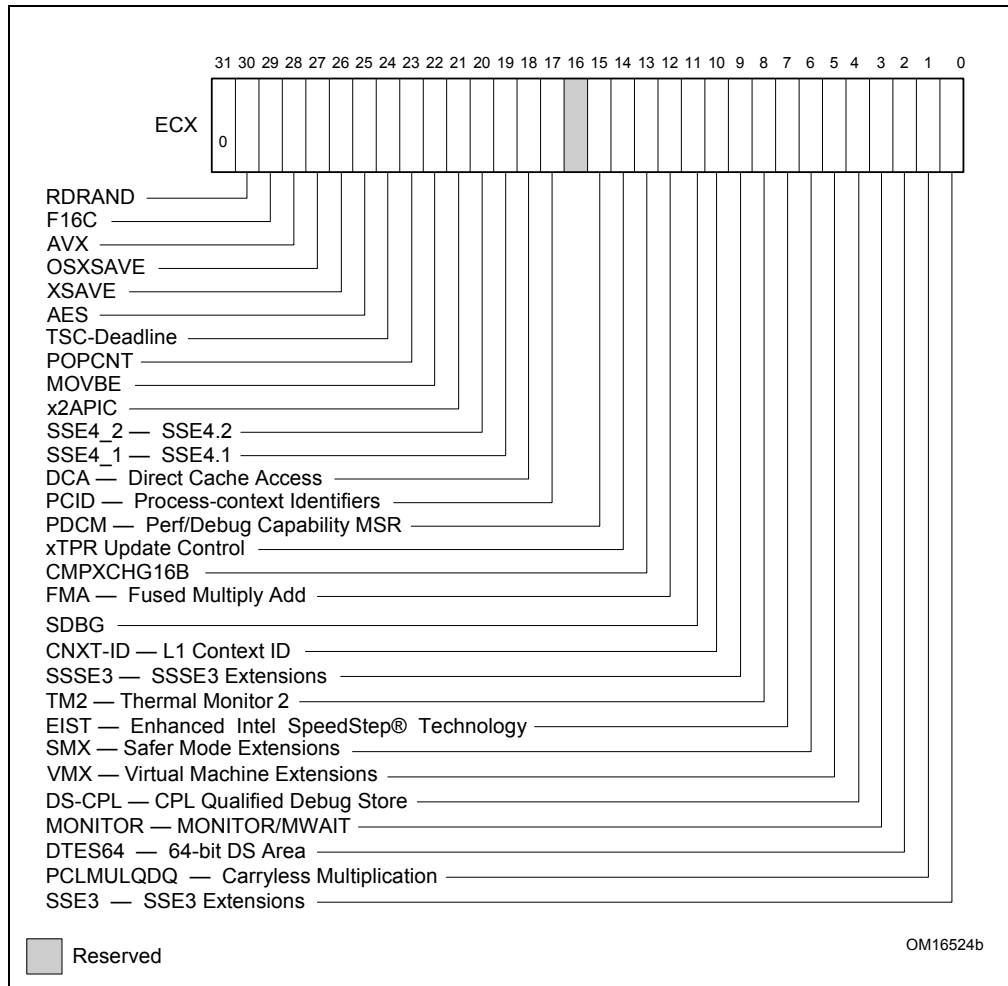


Figure 3-6. Feature Information Returned in the ECX Register

Table 3-19. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	Streaming SIMD Extensions 3 (SSE3). A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	PCLMULQDQ. A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	64-bit DS Area. A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	MONITOR/MWAIT. A value of 1 indicates the processor supports this feature.
4	DS-CPL	CPL Qualified Debug Store. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 5, “Safer Mode Extensions Reference”.
7	EIST	Enhanced Intel SpeedStep® technology. A value of 1 indicates that the processor supports this technology.
8	TM2	Thermal Monitor 2. A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.

Table 3-19. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
10	CNXT-ID	L1 Context ID. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	CMPXCHG16B Available. A value of 1 indicates that the feature is available. See the “CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes” section in this chapter for a description.
14	xTPR Update Control	xTPR Update Control. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	Perfmon and Debug Capability: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	Process-context identifiers. A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor’s local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

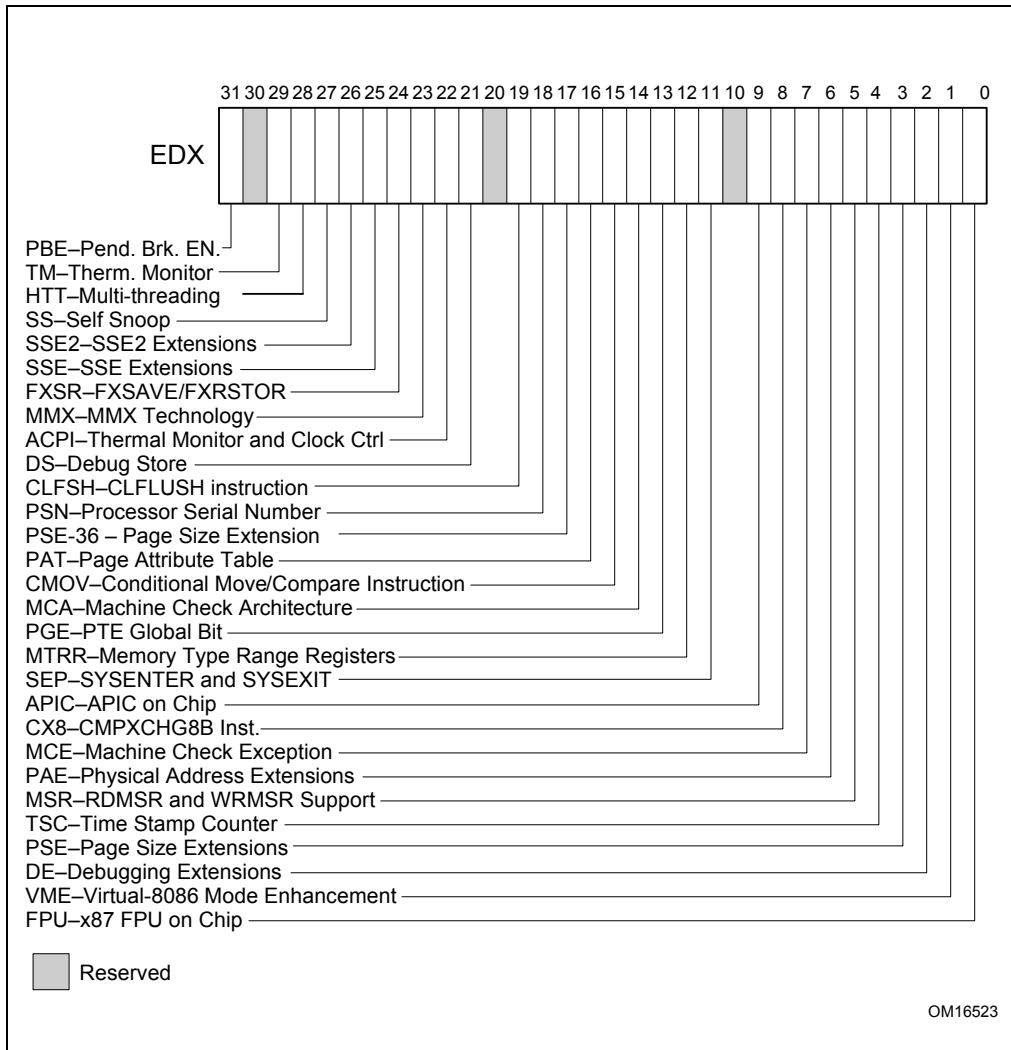


Figure 3-7. Feature Information Returned in the EDX Register

Table 3-20. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	Floating Point Unit On-Chip. The processor contains an x87 FPU.
1	VME	Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	Page Size Extension. Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	Time Stamp Counter. The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	Model Specific Registers RDMSR and WRMSR Instructions. The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	Physical Address Extension. Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	Machine Check Exception. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	CMPXCHG8B Instruction. The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	APIC On-Chip. The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	Memory Type Range Registers. MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	Page Global Bit. The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	Machine Check Architecture. The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	Conditional Move Instructions. The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	Page Attribute Table. Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	36-Bit Page Size Extension. 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	Processor Serial Number. The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	CLFLUSH Instruction. CLFLUSH Instruction is supported.
20	Reserved	Reserved

Table 3-20. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
21	DS	Debug Store. The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 23, “Introduction to Virtual-Machine Extensions,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C</i>).
22	ACPI	Thermal Monitor and Software Controlled Clock Facilities. The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	Intel MMX Technology. The processor supports the Intel MMX technology.
24	FXSR	FXSAVE and FXRSTOR Instructions. The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	SSE. The processor supports the SSE extensions.
26	SSE2	SSE2. The processor supports the SSE2 extensions.
27	SS	Self Snoop. The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	Max APIC IDs reserved field is Valid. A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	Thermal Monitor. The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	Pending Break Enable. The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, the processor returns information about the processor’s internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-21. Table 3-21 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of “cache type” via CPUID leaf 2.

Table 3-21. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-21. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2-MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages, 128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages, 256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 1 GByte pages, 4-way set associative, 4 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Cache	Trace cache: 12 K- μ op, 8-way set associative
71H	Cache	Trace cache: 16 K- μ op, 8-way set associative
72H	Cache	Trace cache: 32 K- μ op, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries

Table 3-21. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GByte pages, 4-way, 16 entries.
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```

EAX    66 5B 50 01H
EBX     0H
ECX     0H
EDX    00 7A 70 00H

```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
 - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
 - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.

- 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
 - 00H - NULL descriptor.
 - 70H - Trace cache: 12 K-μop, 8-way set associative.
 - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sector, 64-byte cache line size.
 - 00H - NULL descriptor.

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-17.

This Cache Size in Bytes

$$= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line_Size} + 1) * (\text{Sets} + 1)$$

$$= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-17.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-17.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-17.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-17), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-17.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-17) is greater than Pn 0. See Table 3-17.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, “Introduction to Virtual-Machine Extensions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is $\geq 0BH$, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-17.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-17.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-17. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

For i = 2 to 62 // sub-leaf 1 is reserved

IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1) // VECTOR is the 64-bit value of EDX:EAX

Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;

FI;

INPUT EAX = 0FH: Returns Platform Quality of Service (PQoS) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 0FH and ECX = n (n ≥ 1 , and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Platform Quality of Service (PQoS) Enforcement Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 10H and ECX = n (n ≥ 1 , and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-17.

When CPUID executes with EAX set to 14H and ECX = n (n > 1 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX and CPUID.(EAX=0DH, ECX= 0H).EDX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-17.

INPUT EAX = 15H: Returns Time Stamp Counter/Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter/Core Crystal Clock. See Table 3-17.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-17.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: “Identification of Earlier IA-32 Processors” in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 3-8 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.

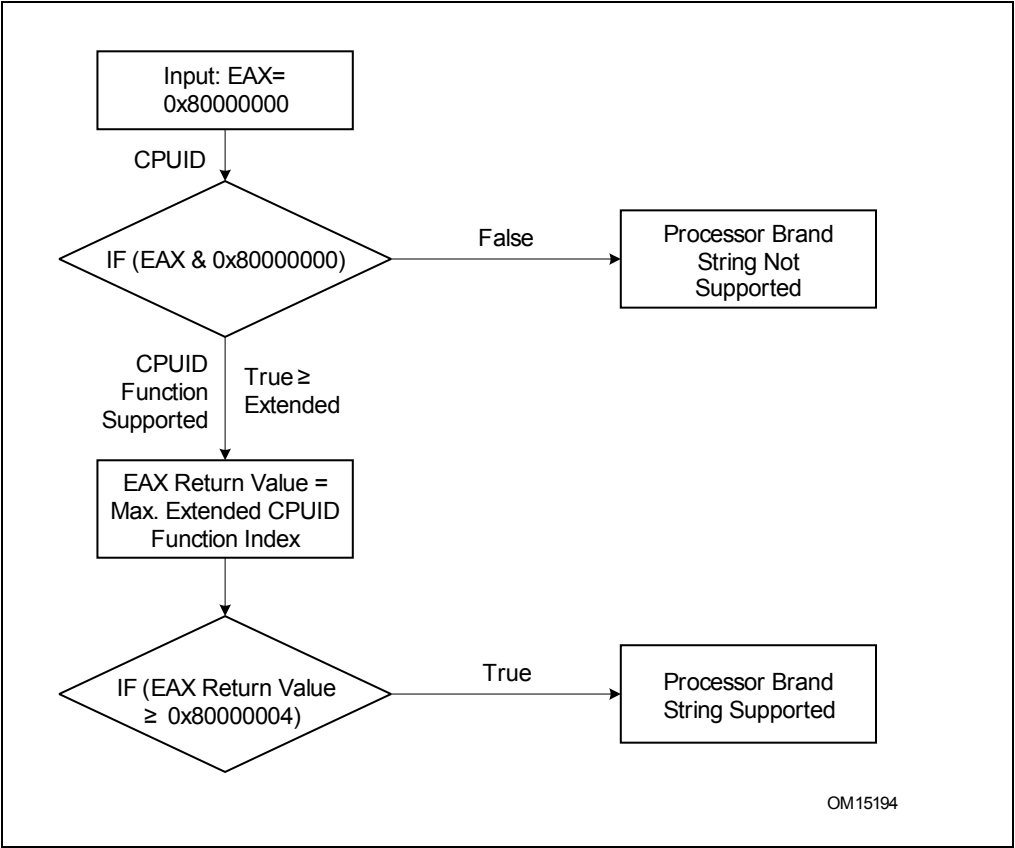


Figure 3-8. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CUID with EAX input of 8000002H through 80000004H. For each input value, CUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-22 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-22. Processor Brand String Returned with Pentium 4 Processor

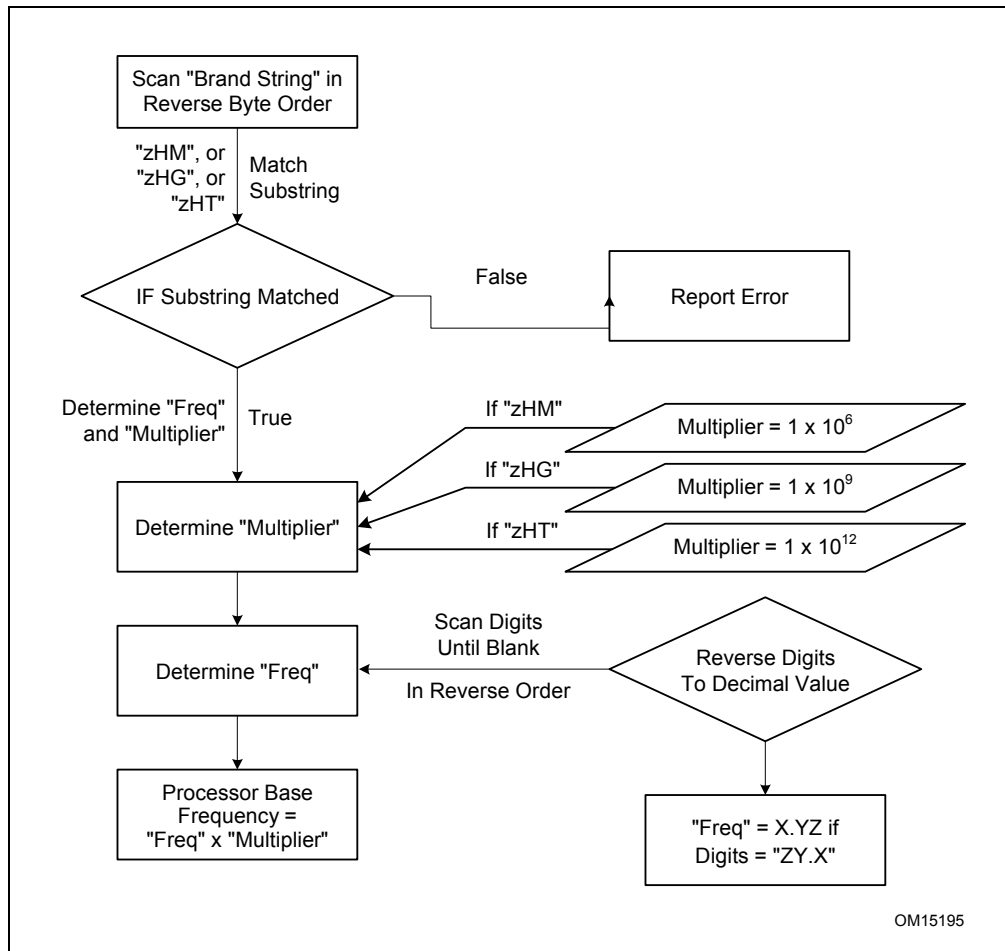
EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	" " " " " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P)R" "itne" "R(mu"

Table 3-22. Processor Brand String Returned with Pentium 4 Processor (Contd.)

80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4)" " UPC" "0051" "\0zHM"
-----------	--	---------------------------------------

Extracting the Processor Frequency from Brand Strings

Figure 3-9 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

**Figure 3-9. Algorithm for Extracting Processor Frequency**

The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not

support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-23 shows brand indices that have identification strings associated with them.

Table 3-23. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor ¹
02H	Intel(R) Pentium(R) III processor ¹
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor ¹
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor ¹
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor ¹
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor ¹
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor ¹
18H – 0FFH	RESERVED

NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

Operation

IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF

EAX = 0:

EAX ← Highest basic function input value understood by CPUID;

EBX ← Vendor identification string;

EDX ← Vendor identification string;

ECX ← Vendor identification string;

BREAK;

EAX = 1H:

EAX[3:0] ← Stepping ID;

EAX[7:4] ← Model;

EAX[11:8] ← Family;
 EAX[13:12] ← Processor type;
 EAX[15:14] ← Reserved;
 EAX[19:16] ← Extended Model;
 EAX[27:20] ← Extended Family;
 EAX[31:28] ← Reserved;
 EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
 EBX[15:8] ← CLFLUSH Line Size;
 EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
 EBX[24:31] ← Initial APIC ID;
 ECX ← Feature flags; (* See Figure 3-6. *)
 EDX ← Feature flags; (* See Figure 3-7. *)

BREAK;

EAX = 2H:

EAX ← Cache and TLB information;
 EBX ← Cache and TLB information;
 ECX ← Cache and TLB information;
 EDX ← Cache and TLB information;

BREAK;

EAX = 3H:

EAX ← Reserved;
 EBX ← Reserved;
 ECX ← ProcessorSerialNumber[31:0];
 (* Pentium III processors only, otherwise reserved. *)
 EDX ← ProcessorSerialNumber[63:32];
 (* Pentium III processors only, otherwise reserved. *)

BREAK

EAX = 4H:

EAX ← Deterministic Cache Parameters Leaf; (* See Table 3-17. *)
 EBX ← Deterministic Cache Parameters Leaf;
 ECX ← Deterministic Cache Parameters Leaf;
 EDX ← Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:

EAX ← MONITOR/MWAIT Leaf; (* See Table 3-17. *)
 EBX ← MONITOR/MWAIT Leaf;
 ECX ← MONITOR/MWAIT Leaf;
 EDX ← MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:

EAX ← Thermal and Power Management Leaf; (* See Table 3-17. *)
 EBX ← Thermal and Power Management Leaf;
 ECX ← Thermal and Power Management Leaf;
 EDX ← Thermal and Power Management Leaf;

BREAK;

EAX = 7H:

EAX ← Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-17. *)
 EBX ← Structured Extended Feature Flags Enumeration Leaf;
 ECX ← Structured Extended Feature Flags Enumeration Leaf;
 EDX ← Structured Extended Feature Flags Enumeration Leaf;

BREAK;

EAX = 8H:

EAX ← Reserved = 0;
 EBX ← Reserved = 0;

```

    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 3-17. *)
    EBX ← Direct Cache Access Information Leaf;
    ECX ← Direct Cache Access Information Leaf;
    EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 3-17. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
EAX = BH:
    EAX ← Extended Topology Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Extended Topology Enumeration Leaf;
    ECX ← Extended Topology Enumeration Leaf;
    EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = DH:
    EAX ← Processor Extended State Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Processor Extended State Enumeration Leaf;
    ECX ← Processor Extended State Enumeration Leaf;
    EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = FH:
    EAX ← Platform Quality of Service Monitoring Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Platform Quality of Service Monitoring Enumeration Leaf;
    ECX ← Platform Quality of Service Monitoring Enumeration Leaf;
    EDX ← Platform Quality of Service Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
    EAX ← Platform Quality of Service Enforcement Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Platform Quality of Service Enforcement Enumeration Leaf;
    ECX ← Platform Quality of Service Enforcement Enumeration Leaf;
    EDX ← Platform Quality of Service Enforcement Enumeration Leaf;
BREAK;
    EAX = 14H:
    EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Intel Processor Trace Enumeration Leaf;

```



```

    ECX ← Intel Processor Trace Enumeration Leaf;
    EDX ← Intel Processor Trace Enumeration Leaf;
BREAK;
EAX = 15H:
    EAX ← Time Stamp Counter/Core Crystal Clock Information Leaf; (* See Table 3-17. *)
    EBX ← Time Stamp Counter/Core Crystal Clock Information Leaf;
    ECX ← Time Stamp Counter/Core Crystal Clock Information Leaf;
    EDX ← Time Stamp Counter/Core Crystal Clock Information Leaf;
BREAK;
EAX = 16H:
    EAX ← Processor Frequency Information Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Processor Frequency Information Enumeration Leaf;
    ECX ← Processor Frequency Information Enumeration Leaf;
    EDX ← Processor Frequency Information Enumeration Leaf;
BREAK;
BREAK;
EAX = 80000000H:
    EAX ← Highest extended function input value understood by CPUID;
    EBX ← Reserved;
    ECX ← Reserved;
    EDX ← Reserved;
BREAK;
EAX = 80000001H:
    EAX ← Reserved;
    EBX ← Reserved;
    ECX ← Extended Feature Bits (* See Table 3-17.*);
    EDX ← Extended Feature Bits (* See Table 3-17. *);
BREAK;
EAX = 80000002H:
    EAX ← Processor Brand String;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000003H:
    EAX ← Processor Brand String, continued;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000004H:
    EAX ← Processor Brand String, continued;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000005H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 80000006H:
    EAX ← Reserved = 0;

```

```

    EBX ← Reserved = 0;
    ECX ← Cache information;
    EDX ← Reserved = 0;
BREAK;
EAX = 80000007H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = Misc Feature Flags;
BREAK;
EAX = 80000008H:
    EAX ← Reserved = Physical Address Size Information;
    EBX ← Reserved = Virtual Address Size Information;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX >= 40000000H and EAX <= 4FFFFFFFH:
DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
    (* If the highest basic information leaf data depend on ECX input value, ECX is honored. *)
    EAX ← Reserved; (* Information returned for highest basic information leaf. *)
    EBX ← Reserved; (* Information returned for highest basic information leaf. *)
    ECX ← Reserved; (* Information returned for highest basic information leaf. *)
    EDX ← Reserved; (* Information returned for highest basic information leaf. *)
BREAK;
ESAC;

```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.</p>
-----	---

CRC32 — Accumulate CRC32 Value

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 38 F0 /r CRC32 r32, r/m8	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 REX 0F 38 F0 /r CRC32 r32, r/m8*	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 0F 38 F1 /r CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 0F 38 F1 /r CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W 0F 38 F0 /r CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W 0F 38 F1 /r CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on r/m64.

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

Operation

Notes:

BIT_REFLECT64: DST[63-0] = SRC[0-63]
 BIT_REFLECT32: DST[31-0] = SRC[0-31]
 BIT_REFLECT16: DST[15-0] = SRC[0-15]
 BIT_REFLECT8: DST[7-0] = SRC[0-7]
 MOD2: Remainder from Polynomial division modulus 2

CRC32 instruction for 64-bit source operand and 64-bit destination operand:

```

TEMP1[63-0] ← BIT_REFLECT64 (SRC[63-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[95-0] ← TEMP1[63-0] « 32
TEMP4[95-0] ← TEMP2[31-0] « 64
TEMP5[95-0] ← TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] ← TEMP5[95-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

```

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

```

TEMP1[31-0] ← BIT_REFLECT32 (SRC[31-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[63-0] ← TEMP1[31-0] « 32
TEMP4[63-0] ← TEMP2[31-0] « 32
TEMP5[63-0] ← TEMP3[63-0] XOR TEMP4[63-0]
TEMP6[31-0] ← TEMP5[63-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

```

TEMP1[15-0] ← BIT_REFLECT16 (SRC[15-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[47-0] ← TEMP1[15-0] « 32
TEMP4[47-0] ← TEMP2[31-0] « 16
TEMP5[47-0] ← TEMP3[47-0] XOR TEMP4[47-0]
TEMP6[31-0] ← TEMP5[47-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

```

TEMP1[7-0] ← BIT_REFLECT8 (SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

```

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

```

TEMP1[7-0] ← BIT_REFLECT8 (SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

```

unsigned int _mm_crc32_u8( unsigned int crc, unsigned char data )
unsigned int _mm_crc32_u16( unsigned int crc, unsigned short data )
unsigned int _mm_crc32_u32( unsigned int crc, unsigned int data )
unsigned __int64 _mm_crc32_u64( unsigned __int64 crc, unsigned __int64 data )

```

SIMD Floating Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

CVTDDQ2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F E6 CVTDDQ2PD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Convert two packed signed doubleword integers from <i>xmm2/m128</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.128.F3.0F.WIG E6 /r VCVTDDQ2PD <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Convert two packed signed doubleword integers from <i>xmm2/mem</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.256.F3.0F.WIG E6 /r VCVTDDQ2PD <i>ymm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed signed doubleword integers from <i>xmm2/mem</i> to four packed double-precision floating-point values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding XMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 128-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

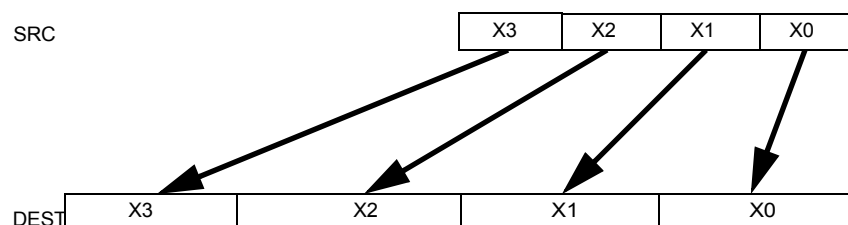


Figure 3-10. CVTDDQ2PD (VEX.256 encoded version)

Operation

CVTDQ2PD (128-bit Legacy SSE version)

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] (unmodified)

VCVTDQ2PD (VEX.128 encoded version)

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] ← 0

VCVTDQ2PD (VEX.256 encoded version)

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
 DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])

Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PD: __m128d _mm_cvtepi32_pd(__m128i a)
 VCVTDQ2PD: __m256d _mm256_cvtepi32_pd (__m128i src)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTDDQ2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 5B /r CVTDDQ2PS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert four packed signed doubleword integers from <i>xmm2/m128</i> to four packed single-precision floating-point values in <i>xmm1</i> .
VEX.128.0F.WIG 5B /r VCVTDQ2PS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed signed doubleword integers from <i>xmm2/mem</i> to four packed single-precision floating-point values in <i>xmm1</i> .
VEX.256.0F.WIG 5B /r VCVTDQ2PS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Convert eight packed signed doubleword integers from <i>ymm2/mem</i> to eight packed single-precision floating-point values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four packed signed doubleword integers in the source operand (second operand) to four packed single-precision floating-point values in the destination operand (first operand).

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding XMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

CVTDDQ2PS (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[VLMAX-1:128] (unmodified)
```

VCVTDQ2PS (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```


VCVTDQ2PS (VEX.256 encoded version)

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
 DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
 DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
 DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
 DEST[159:128] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
 DEST[191:160] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
 DEST[223:192] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
 DEST[255:224] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PS: `__m128 _mm_cvtepi32_ps(__m128i a)`

VCVTDQ2PS: `__m256 _mm256_cvtepi32_ps(__m256i src)`

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTPD2DQ—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> .
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two signed doubleword integers in <i>xmm1</i> .
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ <i>xmm1, ymm2/m256</i>	RM	V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four signed doubleword integers in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

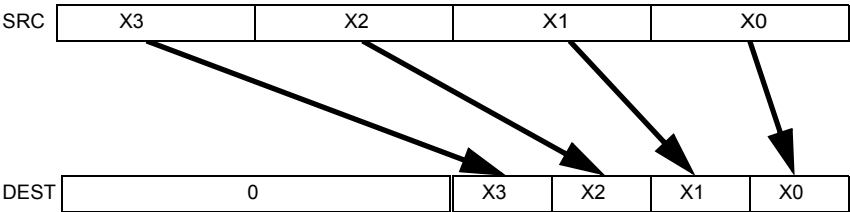


Figure 3-11. VCVTPD2DQ (VEX.256 encoded version)

Operation

CVTPD2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] ← 0
DEST[VLMAX-1:128] (unmodified)

VCVTPD2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[VLMAX-1:64] ← 0

VCVTPD2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192])
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ: __m128i _mm_cvtpd_epi32 (__m128d src)
CVTPD2DQ: __m128i _mm256_cvtpd_epi32 (__m256d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally
#UD If VEX.vvvv ≠ 1111B.

CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2D /r CVTPD2PI <i>mm, xmm/m128</i>	RM	Valid	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PI: `__m64 __mm_cvtpd_pi32(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTPD2PS—Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values in <i>xmm2/m128</i> to two packed single-precision floating-point values in <i>xmm1</i> .
VEX.128.66.0F.WIG 5A /r VCVTPD2PS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two single-precision floating-point values in <i>xmm1</i> .
VEX.256.66.0F.WIG 5A /r VCVTPD2PS <i>xmm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four single-precision floating-point values in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

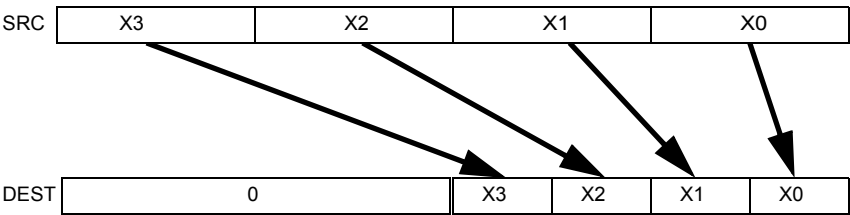


Figure 3-12. VCVTPD2PS (VEX.256 encoded version)

Operation

CVTPD2PS (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
 DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
 DEST[127:64] ← 0
 DEST[VLMAX-1:128] (unmodified)

VCVTPD2PS (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
 DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
 DEST[VLMAX-1:64] ← 0

VCVTPD2PS (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
 DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
 DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
 DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
 DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS: __m128 _mm_cvtpd_ps(__m128d a)
 CVTPD2PS: __m256 _mm256_cvtpd_ps(__m256d a)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /r CVTPI2PD <i>xmm, mm/m64*</i>	RM	Valid	Valid	Convert two packed signed doubleword integers from <i>mm/mem64</i> to two packed double-precision floating-point values in <i>xmm</i> .

NOTES:
*Operation is different for different operand sets; see the Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands *xmm, mm*:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- **For operands *xmm, m64*:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32]);

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PD: __m128d _mm_cvtpi32_pd(__m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-6, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2A /r CVTPI2PS <i>xmm, mm/m64</i>	RM	Valid	Valid	Convert two signed doubleword integers from <i>mm/m64</i> to two single-precision floating-point values in <i>xmm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS: `__m128 __mm_cvtpi32_ps(__m128 a, __m64 b)`

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTPS2DQ—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert four packed single-precision floating-point values from <i>xmm2/m128</i> to four packed signed doubleword integers in <i>xmm1</i> .
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed single precision floating-point values from <i>xmm2/mem</i> to four packed signed doubleword values in <i>xmm1</i> .
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Convert eight packed single precision floating-point values from <i>ymm2/mem</i> to eight packed signed doubleword values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four or eight packed single-precision floating-point values in the source operand to four or eight signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CVTPS2DQ (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[VLMAX-1:128] (unmodified)
```

VCVTPS2DQ (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

VCVTPS2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
 DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
 DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
 DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
 DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
 DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
 DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
 DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2DQ: __m128i _mm_cvtps_epi32(__m128 a)

VCVTPS2DQ: __m256i _mm256_cvtps_epi32 (__m256 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTPS2PD—Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5A /r CVTPS2PD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Convert two packed single-precision floating-point values in <i>xmm2/m64</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.128.OF.WIG 5A /r VCVTPS2PD <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Convert two packed single-precision floating-point values in <i>xmm2/mem</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.256.OF.WIG 5A /r VCVTPS2PD <i>ymm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed single-precision floating-point values in <i>xmm2/mem</i> to four packed double-precision floating-point values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two or four packed single-precision floating-point values in the source operand (second operand) to two or four packed double-precision floating-point values in the destination operand (first operand).

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

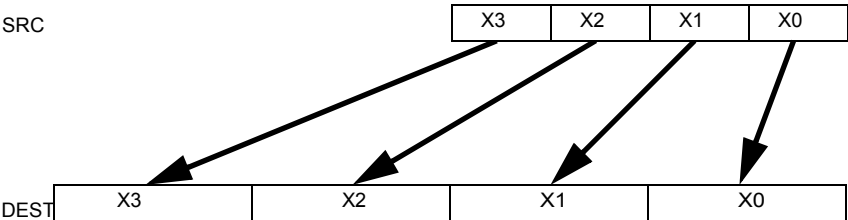


Figure 3-13. CVTPS2PD (VEX.256 encoded version)

Operation

CVTPS2PD (128-bit Legacy SSE version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] (unmodified)

VCVTPS2PD (VEX.128 encoded version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] ← 0

VCVTPS2PD (VEX.256 encoded version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
 DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PD: __m128d _mm_cvtps_pd(__m128 a)
 VCVTPS2PD: __m256d _mm256_cvtps_pd(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX.256 version follows Exception Type 3 without #AC.

Other versions follow Exceptions Type 3; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2D /r CVTPS2PI <i>mm, xmm/m64</i>	RM	Valid	Valid	Convert two packed single-precision floating-point values from <i>xmm/m64</i> to two packed signed doubleword integers in <i>mm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI: `__m64 _mm_cvtps_pi32(__m128 a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTSD2SI—Convert Scalar Double-Precision FP Value to Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer <i>r32</i> .
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed quadword integer sign-extended into <i>r64</i> .
VEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed doubleword integer <i>r32</i> .
VEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64	RM	V/N.E. ¹	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed quadword integer sign-extended into <i>r64</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer64(SRC[63:0]);

ELSE

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);

FI;

Intel C/C++ Compiler Intrinsic Equivalent

`int _mm_cvtsd_si32(__m128d a)`
`__int64 _mm_cvtsd_si64(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally
#UD If VEX.vvvv ≠ 1111B.

CVTSD2SS—Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Convert one double-precision floating-point value in <i>xmm2/m64</i> to one single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5A /r VCVTSD2SS <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Convert one double-precision floating-point value in <i>xmm3/m64</i> to one single-precision floating-point value and merge with high bits in <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a double-precision floating-point value in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are left unchanged. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

CVTSD2SS (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[VLMAX-1:32] Unmodified *)

VCVTSD2SS (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTSD2SS: `__m128 __cvtsd_ss(__m128 a, __m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

CVTSI2SD—Convert Dword Integer to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD <i>xmm</i> , <i>r/m32</i>	RM	V/V	SSE2	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm</i> .
F2 REX.W 0F 2A /r CVTSI2SD <i>xmm</i> , <i>r/m64</i>	RM	V/N.E.	SSE2	Convert one signed quadword integer from <i>r/m64</i> to one double-precision floating-point value in <i>xmm</i> .
VEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD <i>xmm1</i> , <i>xmm2</i> , <i>r/m32</i>	RVM	V/V	AVX	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD <i>xmm1</i> , <i>xmm2</i> , <i>r/m64</i>	RVM	V/N.E. ¹	AVX	Convert one signed quadword integer from <i>r/m64</i> to one double-precision floating-point value in <i>xmm1</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

CVTSI2SD

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);

ELSE

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[VLMAX-1:64] (Unmodified)

VCVTSI2SD

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);

ELSE

DEST[63:0] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);

FI;

DEST[127:64] \leftarrow SRC1[127:64]

DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SD: `__m128d _mm_cvtsi32_sd(__m128d a, int b)`

CVTSI2SD: `__m128d _mm_cvtsi64_sd(__m128d a, __int64 b)`

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 3.

CVTSI2SS—Convert Dword Integer to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS <i>xmm</i> , <i>r/m32</i>	RM	V/V	SSE	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm</i> .
F3 REX.W 0F 2A /r CVTSI2SS <i>xmm</i> , <i>r/m64</i>	RM	V/N.E.	SSE	Convert one signed quadword integer from <i>r/m64</i> to one single-precision floating-point value in <i>xmm</i> .
VEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS <i>xmm1</i> , <i>xmm2</i> , <i>r/m32</i>	RVM	V/V	AVX	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS <i>xmm1</i> , <i>xmm2</i> , <i>r/m64</i>	RVM	V/N.E. ¹	AVX	Convert one signed quadword integer from <i>r/m64</i> to one single-precision floating-point value in <i>xmm1</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

CVTSI2SS (128-bit Legacy SSE version)

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[VLMAX-1:32] (Unmodified)

VCVTSI2SS (VEX.128 encoded version)

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);

ELSE

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[127:32] \leftarrow SRC1[127:32]DEST[VLMAX-1:128] \leftarrow 0**Intel C/C++ Compiler Intrinsic Equivalent**

CVTSI2SS: __m128 _mm_cvtsi32_ss(__m128 a, int b)

CVTSI2SS: __m128 _mm_cvtsi64_ss(__m128 a, __int64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 3.

CVTSS2SD—Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE2	Convert one single-precision floating-point value in <i>xmm2/m32</i> to one double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5A /r VCVTSS2SD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Convert one single-precision floating-point value in <i>xmm3/m32</i> to one double-precision floating-point value and merge with high bits of <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is left unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8–XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX–1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX–1:128) of the destination YMM register are zeroed.

Operation

CVTSS2SD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[VLMAX-1:64] (Unmodified)
```

VCVTSS2SD (VEX.128 encoded version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTSS2SD:    __m128d _mm_cvtss_sd(__m128d a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Exceptions Type 3.

CVTSS2SI—Convert Scalar Single-Precision FP Value to Dword Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm/m32	RM	V/V	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> .
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed quadword integer in <i>r64</i> .
VEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed doubleword integer in <i>r32</i> .
VEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32	RM	V/N.E. ¹	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed quadword integer in <i>r64</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
IF 64-bit Mode and OperandSize = 64
    THEN
        DEST[64:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
    ELSE
        DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

`int _mm_cvtss_si32(__m128d a)`

`__int64 _mm_cvtss_si64(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTTPD2DQ—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> using truncation.
VEX.128.66.0F.WIG E6 /r VCVTPD2DQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two signed doubleword integers in <i>xmm1</i> using truncation.
VEX.256.66.0F.WIG E6 /r VCVTPD2DQ <i>xmm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four signed doubleword integers in <i>xmm1</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two or four packed double-precision floating-point values in the source operand (second operand) to two or four packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

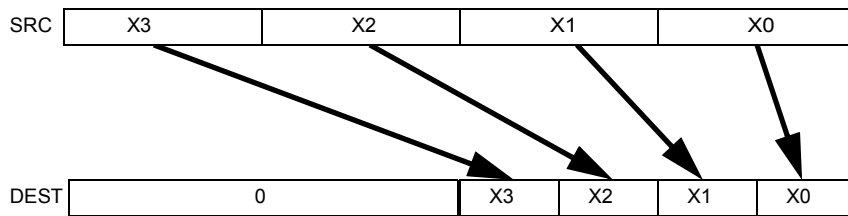


Figure 3-14. VCVTTPD2DQ (VEX.256 encoded version)

Operation

CVTTPD2DQ (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[63:0])$
 $\text{DEST}[63:32] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[127:64])$
 $\text{DEST}[127:64] \leftarrow 0$
 $\text{DEST}[\text{VLMAX}-1:128]$ (unmodified)

VCVTTPD2DQ (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[63:0])$
 $\text{DEST}[63:32] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[127:64])$
 $\text{DEST}[\text{VLMAX}-1:64] \leftarrow 0$

VCVTTPD2DQ (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[63:0])$
 $\text{DEST}[63:32] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[127:64])$
 $\text{DEST}[95:64] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[191:128])$
 $\text{DEST}[127:96] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer_Truncate}(\text{SRC}[255:192])$
 $\text{DEST}[255:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD2DQ: `__m128i _mm_cvttpd_epi32(__m128d a)`

VCVTTPD2DQ: `__m128i _mm256_cvttpd_epi32(__m256d src)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv \neq 1111B.

CVTTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTTPD2PI <i>mm</i> , <i>xmm/m128</i>	RM	Valid	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32_
               Truncate(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI: `__m64 _mm_cvttpd_pi32(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTTPS2DQ—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Convert four single-precision floating-point values from <i>xmm2/m128</i> to four signed doubleword integers in <i>xmm1</i> using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Convert four packed single precision floating-point values from <i>xmm2/mem</i> to four packed signed doubleword values in <i>xmm1</i> using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Convert eight packed single precision floating-point values from <i>ymm2/mem</i> to eight packed signed doubleword values in <i>ymm1</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four or eight packed single-precision floating-point values in the source operand to four or eight signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CVTTPS2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])

DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])

DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])

DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])

DEST[VLMAX-1:128] (unmodified)

VCVTTPS2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
 DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
 DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
 DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
 DEST[VLMAX-1:128] ← 0

VCVTTPS2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
 DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
 DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
 DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
 DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
 DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
 DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
 DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2DQ: `__m128i _mm_cvttps_epi32(__m128 a)`

VCVTTPS2DQ: `__m256i _mm256_cvttps_epi32 (__m256 a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 2C /r CVTTPS2PI <i>mm, xmm/m64</i>	RM	Valid	Valid	Convert two single-precision floating-point values from <i>xmm/m64</i> to two signed doubleword signed integers in <i>mm</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI: `__m64 _mm_cvttps_pi32(__m128 a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTTSD2SI—Convert with Truncation Scalar Double-Precision FP Value to Signed Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm/m64	RM	V/N.E.	SSE2	Convert one double precision floating-point value from <i>xmm/m64</i> to one signed quadword integer in <i>r64</i> using truncation.
VEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from <i>xmm1/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.
VEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64	RM	V/N.E. ¹	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed quadword integer in <i>r64</i> using truncation.

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_
Integer64_Truncate(SRC[63:0]);

ELSE

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_
Integer32_Truncate(SRC[63:0]);

FI;

Intel C/C++ Compiler Intrinsic Equivalent

`int _mm_cvttssd_si32(__m128d a)`

`__int64 _mm_cvttssd_si64(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv ≠ 1111B.

CVTTSS2SI—Convert with Truncation Scalar Single-Precision FP Value to Dword Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm/m32	RM	V/V	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed quadword integer in <i>r64</i> using truncation.
VEX.LIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.
VEX.LIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32	RM	V/N.E. ¹	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed quadword integer in <i>r64</i> using truncation.

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_
                    Integer_Truncate(SRC[31:0]);
    ELSE
        DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_
                    Integer_Truncate(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

`int _mm_cvtss_si32(__m128d a)`

`__int64 _mm_cvtss_si64(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv ≠ 1111B.

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
99	CWD	NP	Valid	Valid	DX:AX ← sign-extend of AX.
99	CDQ	NP	Valid	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO	NP	Valid	N.E.	RDX:RAX ← sign-extend of RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

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

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF OperandSize = 16 (* CWD instruction *)
    THEN
        DX ← SignExtend(AX);
    ELSE IF OperandSize = 32 (* CDQ instruction *)
        EDX ← SignExtend(EAX); FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
        RDX ← SignExtend(RAX); FI;
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
27	DAA	NP	Invalid	Valid	Decimal adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```

IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ← AL;
        old_CF ← CF;
        CF ← 0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL ← AL + 6;
                CF ← old_CF or (Carry from AL ← AL + 6);
                AF ← 1;
            ELSE
                AF ← 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ← AL + 60H;
                CF ← 1;
            ELSE
                CF ← 0;
        FI;
    FI;

```

Example

```

ADD    AL, BL    Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX
                  After: AL=AEH BL=35H EFLAGS(OSZAPC)=1 10000
DAA     Before: AL=AEH BL=35H EFLAGS(OSZAPC)=1 10000
                  After: AL=14H BL=35H EFLAGS(OSZAPC)=X00111
DAA     Before: AL=2EH BL=35H EFLAGS(OSZAPC)=1 10000
                  After: AL=34H BL=35H EFLAGS(OSZAPC)=X00101

```

Flags Affected

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

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
2F	DAS	NP	Invalid	Valid	Decimal adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

old_AL \leftarrow AL;

old_CF \leftarrow CF;

CF \leftarrow 0;

IF (((AL AND 0FH) > 9) or AF = 1)

THEN

AL \leftarrow AL - 6;

CF \leftarrow old_CF or (Borrow from AL \leftarrow AL - 6);

AF \leftarrow 1;

ELSE

AF \leftarrow 0;

FI;

IF ((old_AL > 99H) or (old_CF = 1))

THEN

AL \leftarrow AL - 60H;

CF \leftarrow 1;

FI;

FI;

Example

```

SUB  AL, BL   Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
                After: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
DAA                                Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
                After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111

```

Flags Affected

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

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

DEC—Decrement by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /1	DEC <i>r/m8</i>	M	Valid	Valid	Decrement <i>r/m8</i> by 1.
REX + FE /1	DEC <i>r/m8</i> *	M	Valid	N.E.	Decrement <i>r/m8</i> by 1.
FF /1	DEC <i>r/m16</i>	M	Valid	Valid	Decrement <i>r/m16</i> by 1.
FF /1	DEC <i>r/m32</i>	M	Valid	Valid	Decrement <i>r/m32</i> by 1.
REX.W + FF /1	DEC <i>r/m64</i>	M	Valid	N.E.	Decrement <i>r/m64</i> by 1.
48+rw	DEC <i>r16</i>	O	N.E.	Valid	Decrement <i>r16</i> by 1.
48+rd	DEC <i>r32</i>	O	N.E.	Valid	Decrement <i>r32</i> by 1.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (<i>r, w</i>)	NA	NA	NA
O	opcode + rd (<i>r, w</i>)	NA	NA	NA

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.

In 64-bit mode, DEC *r16* and DEC *r32* are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

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 non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

DIV—Unsigned Divide

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /6	DIV <i>r/m8</i>	M	Valid	Valid	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV <i>r/m8</i> *	M	Valid	N.E.	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV <i>r/m16</i>	M	Valid	Valid	Unsigned divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV <i>r/m32</i>	M	Valid	Valid	Unsigned divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /6	DIV <i>r/m64</i>	M	Valid	N.E.	Unsigned divide RDX:RAX by <i>r/m64</i> , with result stored in RAX ← Quotient, RDX ← Remainder.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

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

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-24.

Table 3-24. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	<i>r/m8</i>	AL	AH	255
Doubleword/word	DX:AX	<i>r/m16</i>	AX	DX	65,535
Quadword/doubleword	EDX:EAX	<i>r/m32</i>	EAX	EDX	2 ³² – 1
Doublequadword/quadword	RDX:RAX	<i>r/m64</i>	RAX	RDX	2 ⁶⁴ – 1

Operation

```

IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp ← AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX MOD SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
                ELSE
                    AX ← temp;
                    DX ← DX:AX MOD SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ← EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
                ELSE
                    EAX ← temp;
                    EDX ← EDX:EAX MOD SRC;
            FI;
        FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
                ELSE
                    RAX ← temp;
                    RDX ← RDX:RAX MOD SRC;
            FI;
        FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#DE	<p>If the source operand (divisor) is 0</p> <p>If the quotient is too large for the designated register.</p>
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>

#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
#GP	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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
#GP(0)	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.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#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.
#UD	If the LOCK prefix is used.

DIVPD—Divide Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5E /r DIVPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Divide packed double-precision floating-point values in <i>xmm1</i> by packed double-precision floating-point values <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Divide packed double-precision floating-point values in <i>xmm2</i> by packed double-precision floating-point values in <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Divide packed double-precision floating-point values in <i>ymm2</i> by packed double-precision floating-point values in <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD divide of the two or four packed double-precision floating-point values in the first source operand by the two or four packed double-precision floating-point values in the second source operand. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

DIVPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[VLMAX-1:128] (Unmodified)

VDIVPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[VLMAX-1:128] ← 0

VDIVPD (VEX.256 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] / \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] / \text{SRC2}[127:64]$$

$$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] / \text{SRC2}[191:128]$$

$$\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] / \text{SRC2}[255:192]$$
Intel C/C++ Compiler Intrinsic Equivalent

DIVPD: `__m128d _mm_div_pd(__m128d a, __m128d b)`

VDIVPD: `__m256d _mm256_div_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5E /r DIVPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Divide packed single-precision floating-point values in <i>xmm1</i> by packed single-precision floating-point values <i>xmm2/m128</i> .
VEX.NDS.128.OF.WIG 5E /r VDIVPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Divide packed single-precision floating-point values in <i>xmm2</i> by packed double-precision floating-point values in <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 5E /r VDIVPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Divide packed single-precision floating-point values in <i>ymm2</i> by packed double-precision floating-point values in <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD divide of the four or eight packed single-precision floating-point values in the first source operand by the four or eight packed single-precision floating-point values in the second source operand. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

DIVPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
 DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
 DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
 DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
 DEST[VLMAX-1:128] (Unmodified)

VDIVPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
 DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
 DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
 DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
 DEST[VLMAX-1:128] ← 0

VDIVPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] / \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] / \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] / \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] / \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] / \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] / \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] / \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] / \text{SRC2}[255:224]$.

Intel C/C++ Compiler Intrinsic Equivalent

DIVPS: `__m128 _mm_div_ps(__m128 a, __m128 b)`
 VDIVPS: `__m256 _mm256_div_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

DIVSD—Divide Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD <i>xmm1</i> , <i>xmm2/mem64</i>	RM	V/V	SSE2	Divide low double-precision floating-point value in <i>xmm1</i> by low double-precision floating-point value in <i>xmm2/mem64</i> .
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/mem64</i>	RVM	V/V	AVX	Divide low double-precision floating point values in <i>xmm2</i> by low double precision floating-point value in <i>xmm3/mem64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination hyperons are XMM registers. The high quadword of the destination operand is copied from the high quadword of the first source operand. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

DIVSD (128-bit Legacy SSE version)

$DEST[63:0] \leftarrow DEST[63:0] / SRC[63:0]$

$DEST[VLMAX-1:64]$ (Unmodified)

VDIVSD (VEX.128 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] / SRC2[63:0]$

$DEST[127:64] \leftarrow SRC1[127:64]$

$DEST[VLMAX-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

DIVSD: `__m128d _mm_div_sd (m128d a, m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Divide low single-precision floating-point value in <i>xmm1</i> by low single-precision floating-point value in <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Divide low single-precision floating point value in <i>xmm2</i> by low single precision floating-point value in <i>xmm3/m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination are copied from the same dwords of the first source operand. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

DIVSS (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow DEST[31:0] / SRC[31:0]$

$DEST[VLMAX-1:32]$ (Unmodified)

VDIVSS (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] / SRC2[31:0]$

$DEST[127:32] \leftarrow SRC1[127:32]$

$DEST[VLMAX-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

DIVSS: `__m128 _mm_div_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

DPPD — Dot Product of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed DP floating-point values from <i>xmm1</i> with packed DP floating-point values from <i>xmm2</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 41 /r ib VDPPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Selectively multiply packed DP floating-point values from <i>xmm2</i> with packed DP floating-point values from <i>xmm3</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Conditionally multiplies the packed double-precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The two resulting double-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPD follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] = 1)
    THEN Temp1[63:0] ← DEST[63:0] * SRC[63:0]; // update SIMD exception flags
    ELSE Temp1[63:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] ← +0.0; FI;
/* if unmasked exception reported, execute exception handler*/
```

```
Temp2[63:0] ← Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
```

```
IF (imm8[0] = 1)
    THEN DEST[63:0] ← Temp2[63:0];
    ELSE DEST[63:0] ← +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← Temp2[63:0];
    ELSE DEST[127:64] ← +0.0; FI;
```

DPPD (128-bit Legacy SSE version)

```
DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)
```

VDPPD (VEX.128 encoded version)

```
DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] ← 0
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

DPPD: `__m128d _mm_dp_pd (__m128d a, __m128d b, const int mask);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.L = 1.

DPPS — Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from <i>xmm1</i> with packed SP floating-point values from <i>xmm2</i> , add and selectively store the packed SP floating-point values or zero values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Multiply packed SP floating point values from <i>xmm1</i> with packed SP floating point values from <i>xmm2/mem</i> selectively add and store to <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm2</i> with packed SP floating point values from <i>ymm3/mem</i> , selectively add pairs of elements and store to <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single-precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in Imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The four resulting single-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPS follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

DP_primitive (SRC1, SRC2)

```

IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0]; // update SIMD exception flags
    ELSE Temp1[31:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32]; // update SIMD exception flags
    ELSE Temp1[63:32] ← +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64]; // update SIMD exception flags
    ELSE Temp1[95:64] ← +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96]; // update SIMD exception flags
    ELSE Temp1[127:96] ← +0.0; FI;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)
    THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; FI;

```

DPPS (128-bit Legacy SSE version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)

```

VDPPS (VEX.128 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] ← 0

```

VDPPS (VEX.256 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← DP_Primitive(SRC1[255:128], SRC2[255:128]);

```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

(V)DPPS: `__m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);`

VDPPS: `__m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

Other Exceptions

See Exceptions Type 2.

EMMS—Empty MMX Technology State

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 77	EMMS	NP	Valid	Valid	Set the x87 FPU tag word to empty.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
x87FPUTagWord ← FFFFH;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_empty()
```

Flags Affected

None.

Protected Mode Exceptions

#UD If CR0.EM[bit 2] = 1.
 #NM If CR0.TS[bit 3] = 1.
 #MF If there is a pending FPU exception.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C8 <i>iw</i> 00	ENTER <i>imm16</i> , 0	II	Valid	Valid	Create a stack frame for a procedure.
C8 <i>iw</i> 01	ENTER <i>imm16</i> , 1	II	Valid	Valid	Create a nested stack frame for a procedure.
C8 <i>iw</i> ib	ENTER <i>imm16</i> , <i>imm8</i>	II	Valid	Valid	Create a nested stack frame for a procedure.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	<i>iw</i>	<i>imm8</i>	NA	NA

Description

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

The stack-size attribute determines whether the BP (16 bits), EBP (32 bits), or RBP (64 bits) register specifies the current frame pointer and whether SP (16 bits), ESP (32 bits), or RSP (64 bits) specifies the stack pointer. In 64-bit mode, stack-size attribute is always 64-bits.

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

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded.

Operation

NestingLevel ← NestingLevel MOD 32

IF 64-Bit Mode (StackSize = 64)

THEN

Push(RBP);

FrameTemp ← RSP;

ELSE IF StackSize = 32

THEN

Push(EBP);

FrameTemp ← ESP; FI;

ELSE (* StackSize = 16 *)

Push(BP);

FrameTemp ← SP;

FI;

IF NestingLevel = 0

```

    THEN GOTO CONTINUE;
FI;

IF (NestingLevel > 1)
    THEN FOR i ← 1 to (NestingLevel - 1)
        DO
            IF 64-Bit Mode (StackSize = 64)
                THEN
                    RBP ← RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            EBP ← EBP - 4;
                            Push([EBP]); (* Doubleword push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 4;
                            Push([BP]); (* Doubleword push *)
                        FI;
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 2;
                            Push([BP]); (* Word push *)
                        FI;
                    FI;
                FI;
            OD;
        FI;

    IF 64-Bit Mode (StackSize = 64)
        THEN
            Push(FrameTemp); (* Quadword push *)
        ELSE IF OperandSize = 32
            THEN
                Push(FrameTemp); FI; (* Doubleword push *)
        ELSE (* OperandSize = 16 *)
            Push(FrameTemp); (* Word push *)
        FI;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP ← FrameTemp;
        RSP ← RSP - Size;
    ELSE IF StackSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP - Size; FI;
    ELSE (* StackSize = 16 *)
        BP ← FrameTemp;
        SP ← SP - Size;

```

FI;

END;

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

EXTRACTPS — Extract Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS <i>reg/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a single-precision floating-point value from <i>xmm2</i> at the source offset specified by <i>imm8</i> and store the result to <i>reg</i> or <i>m32</i> . The upper 32 bits of <i>r64</i> is zeroed if <i>reg</i> is <i>r64</i> .
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS <i>r/m32, xmm1, imm8</i>	MRI	V/V	AVX	Extract one single-precision floating-point value from <i>xmm1</i> at the offset specified by <i>imm8</i> and store the result in <i>reg</i> or <i>m32</i> . Zero extend the results in 64-bit register if applicable.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from *imm8*. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.128 encoded version: When VEX.128.66.0F3A.W1 17 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits. VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

The source register is an XMM register. *Imm8[1:0]* determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation**EXTRACTPS (128-bit Legacy SSE version)**

$SRC_OFFSET \leftarrow IMM8[1:0]$

IF (64-Bit Mode and DEST is register)

$DEST[31:0] \leftarrow (SRC[127:0] \gg (SRC_OFFSET * 32)) \text{ AND } 0FFFFFFFh$

$DEST[63:32] \leftarrow 0$

ELSE

$DEST[31:0] \leftarrow (SRC[127:0] \gg (SRC_OFFSET * 32)) \text{ AND } 0FFFFFFFh$

FI

VEXTRACTPS (VEX.128 encoded version)

SRC_OFFSET \leftarrow IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] \leftarrow (SRC[127:0] \gg (SRC_OFFSET*32)) AND 0FFFFFFFh

DEST[63:32] \leftarrow 0

ELSE

DEST[31:0] \leftarrow (SRC[127:0] \gg (SRC_OFFSET*32)) AND 0FFFFFFFh

FI

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS: _mm_extractmem_ps (float *dest, __m128 a, const int idx);

EXTRACTPS: __m128 _mm_extract_ps (__m128 a, const int idx);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.

F2XM1—Compute 2^x-1

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with $(2^{\text{ST}(0)} - 1)$.

Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range -1.0 to $+1.0$. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-25. Results Obtained from F2XM1

ST(0) SRC	ST(0) DEST
-1.0 to -0	-0.5 to -0
-0	-0
$+0$	$+0$
$+0$ to $+1.0$	$+0$ to 1.0

Values other than 2 can be exponentiated using the following formula:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$\text{ST}(0) \leftarrow (2^{\text{ST}(0)} - 1);$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FABS—Absolute Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E1	FABS	Valid	Valid	Replace ST with its absolute value.

Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

Table 3-26. Results Obtained from FABS

ST(0) SRC	ST(0) DEST
− ∞	+ ∞
− F	+ F
− 0	+ 0
+ 0	+ 0
+ F	+ F
+ ∞	+ ∞
NaN	NaN

NOTES:

F Means finite floating-point value.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

$ST(0) \leftarrow |ST(0)|;$

FPU Flags Affected

- C1 Set to 0.
- C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FADD/FADDP/FIADD—Add

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /0	FADD <i>m32fp</i>	Valid	Valid	Add <i>m32fp</i> to ST(0) and store result in ST(0).
DC /0	FADD <i>m64fp</i>	Valid	Valid	Add <i>m64fp</i> to ST(0) and store result in ST(0).
D8 C0+i	FADD ST(0), ST(i)	Valid	Valid	Add ST(0) to ST(i) and store result in ST(0).
DC C0+i	FADD ST(i), ST(0)	Valid	Valid	Add ST(i) to ST(0) and store result in ST(i).
DE C0+i	FADDP ST(i), ST(0)	Valid	Valid	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
DE C1	FADDP	Valid	Valid	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
DA /0	FIADD <i>m32int</i>	Valid	Valid	Add <i>m32int</i> to ST(0) and store result in ST(0).
DE /0	FIADD <i>m16int</i>	Valid	Valid	Add <i>m16int</i> to ST(0) and store result in ST(0).

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is -0 . When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-27.

Table 3-27. FADD/FADDP/FIADD Results

SRC	DEST							
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$ or $-I$	$-\infty$	$-F$	SRC	SRC	$\pm F$ or ± 0	$+\infty$	NaN
	-0	$-\infty$	DEST	-0	± 0	DEST	$+\infty$	NaN
	$+0$	$-\infty$	DEST	± 0	$+0$	DEST	$+\infty$	NaN
	$+F$ or $+I$	$-\infty$	$\pm F$ or ± 0	SRC	SRC	$+F$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF Instruction = FIADD

THEN

DEST \leftarrow DEST + ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST \leftarrow DEST + SRC;

FI;

IF Instruction = FADDP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.
Operands are infinities of unlike sign.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /4	FBLD <i>m80dec</i>	Valid	Valid	Convert BCD value to floating-point and push onto the FPU stack.

Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of -0 .

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$TOP \leftarrow TOP - 1$;

$ST(0) \leftarrow \text{ConvertToDoubleExtendedPrecisionFP}(\text{SRC})$;

FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack overflow occurred.

Protected Mode Exceptions

#GP(0) 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.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD If the LOCK prefix is used.

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.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /6	FBSTP m80bcd	Valid	Valid	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

Table 3-28. FBSTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	– D
$-1 < F < -0$	**
– 0	– 0
+ 0	+ 0
$+0 < F < +1$	**
$F \geq +1$	+ D
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

* Indicates floating-point invalid-operation (#IA) exception.

** ± 0 or ± 1 , depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞ , SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$DEST \leftarrow BCD(ST(0));$

PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value that exceeds 18 BCD digits in length. Source operand is an SNaN, QNaN, $\pm\infty$, or in an unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that points to a non-writable 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FCHS—Change Sign

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E0	FCHS	Valid	Valid	Complements sign of ST(0).

Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-29. FCHS Results

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
$-F$	$+F$
-0	$+0$
$+0$	-0
$+F$	$-F$
$+\infty$	$-\infty$
NaN	NaN

NOTES:

* F means finite floating-point value.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$\text{SignBit}(\text{ST}(0)) \leftarrow \text{NOT}(\text{SignBit}(\text{ST}(0)))$;

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCLEX/FNCLEX—Clear Exceptions

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DB E2	FCLEX	Valid	Valid	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX*	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
FPUStatusWord[0:7] ← 0;
FPUStatusWord[15] ← 0;
```

FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

```
#NM      CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD      If the LOCK prefix is used.
```

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCMOVcc—Floating-Point Conditional Move

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode*	Description
DA C0+i	FCMOVB ST(0), ST(i)	Valid	Valid	Move if below (CF=1).
DA C8+i	FCMOVE ST(0), ST(i)	Valid	Valid	Move if equal (ZF=1).
DA D0+i	FCMOVBE ST(0), ST(i)	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
DA D8+i	FCMOVU ST(0), ST(i)	Valid	Valid	Move if unordered (PF=1).
DB C0+i	FCMOVNB ST(0), ST(i)	Valid	Valid	Move if not below (CF=0).
DB C8+i	FCMOVNE ST(0), ST(i)	Valid	Valid	Move if not equal (ZF=0).
DB D0+i	FCMOVNBE ST(0), ST(i)	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
DB D8+i	FCMOVNU ST(0), ST(i)	Valid	Valid	Move if not unordered (PF=0).

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic is given in the Description column above and in Chapter 8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

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

A processor may not support the FCMOVcc instructions. Software can check if the FCMOVcc instructions are supported by checking the processor's feature information with the CPUID instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOVcc instructions are supported.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
IF condition TRUE
    THEN ST(0) ← ST(i);
FI;
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

Integer Flags Affected

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /2	FCOM <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> .
DC /2	FCOM <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> .
D8 D0+i	FCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
D8 D1	FCOM	Valid	Valid	Compare ST(0) with ST(1).
D8 /3	FCOMP <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> and pop register stack.
DC /3	FCOMP <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> and pop register stack.
D8 D8+i	FCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that -0.0 is equal to $+0.0$.

Table 3-30. FCOM/FCOMP/FCOMPP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered*	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

CASE (relation of operands) OF

ST > SRC: C3, C2, C0 ← 000;

ST < SRC: C3, C2, C0 ← 001;

ST = SRC: C3, C2, C0 ← 100;

ESAC;

IF ST(0) or SRC = NaN or unsupported format

THEN

#IA

IF FPUControlWord.IM = 1

THEN

C3, C2, C0 ← 111;

FI;

FI;

IF Instruction = FCOMP

THEN

PopRegisterStack;

FI;

IF Instruction = FCOMPP

THEN

PopRegisterStack;

PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 See table on previous page.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA One or both operands are NaN values or have unsupported formats.
Register is marked empty.

#D One or both operands are denormal values.

Protected Mode Exceptions

#GP(0) 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.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#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.

#UD If the LOCK prefix is used.

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.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
 #SS(0) If a memory operand effective address is outside the SS segment limit.
 #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #PF(fault-code) If a page fault occurs.
 #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
 #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
 #GP(0) If the memory address is in a non-canonical form.
 #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #MF If there is a pending x87 FPU exception.
 #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.
 #UD If the LOCK prefix is used.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DB F0+i	FCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i) and set status flags accordingly.
DF F0+i	FCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.
DB E8+i	FUCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.
DF E8+i	FUCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.

Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

Table 3-31. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results

Comparison Results*	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered**	1	1	1

NOTES:

* See the IA-32 Architecture Compatibility section below.

** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

CASE (relation of operands) OF

ST(0) > ST(i): ZF, PF, CF ← 000;

ST(0) < ST(i): ZF, PF, CF ← 001;

ST(0) = ST(i): ZF, PF, CF ← 100;

ESAC;

IF Instruction is FCOMI or FCOMIP

THEN

IF ST(0) or ST(i) = NaN or unsupported format

THEN

#IA

IF FPUControlWord.IM = 1

THEN

ZF, PF, CF ← 111;

FI;

FI;

FI;

IF Instruction is FUCOMI or FUCOMIP

THEN

IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format

THEN

ZF, PF, CF ← 111;

ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)

#IA;

IF FPUControlWord.IM = 1

THEN

ZF, PF, CF ← 111;

FI;

FI;

FI;

IF Instruction is FCOMIP or FUCOMIP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 Not affected.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA (FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats.

(FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCOS— Cosine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FF	FCOS	Valid	Valid	Replace ST(0) with its approximate cosine.

Description

Computes the approximate cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers.

Table 3-32. FCOS Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
$-F$	-1 to $+1$
-0	$+1$
$+0$	$+1$
$+F$	-1 to $+1$
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF |ST(0)| < 263
THEN
    C2 ← 0;
    ST(0) ← FCOS(ST(0)); // approximation of cosine
ELSE (* Source operand is out-of-range *)
    C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise. Undefined if C2 is 1.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.

C0, C3 Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F6	FDECSTP	Valid	Valid	Decrement TOP field in FPU status word.

Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF TOP = 0
    THEN TOP ← 7;
    ELSE TOP ← TOP - 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #MF If there is a pending x87 FPU exception.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /6	FDIV <i>m32fp</i>	Valid	Valid	Divide ST(0) by <i>m32fp</i> and store result in ST(0).
DC /6	FDIV <i>m64fp</i>	Valid	Valid	Divide ST(0) by <i>m64fp</i> and store result in ST(0).
D8 F0+i	FDIV ST(0), ST(i)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(0).
DC F8+i	FDIV ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(i).
DE F8+i	FDIVP ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE F9	FDIVP	Valid	Valid	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /6	FIDIV <i>m32int</i>	Valid	Valid	Divide ST(0) by <i>m32int</i> and store result in ST(0).
DE /6	FIDIV <i>m16int</i>	Valid	Valid	Divide ST(0) by <i>m16int</i> and store result in ST(0).

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-33. FDIV/FDIVP/FIDIV Results

		DEST						
SRC		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	$+0$	$+0$	-0	-0	*	NaN
	$-F$	$+\infty$	$+F$	$+0$	-0	$-F$	$-\infty$	NaN
	$-I$	$+\infty$	$+F$	$+0$	-0	$-F$	$-\infty$	NaN
	-0	$+\infty$	**	*	*	**	$-\infty$	NaN
	$+0$	$-\infty$	**	*	*	**	$+\infty$	NaN
	$+I$	$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	-0	-0	$+0$	$+0$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF SRC = 0
    THEN
        #Z;
    ELSE
        IF Instruction is FIDIV
            THEN
                DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* Source operand is floating-point value *)
                DEST ← DEST / SRC;
        FI;
    FI;

IF Instruction = FDIVP
    THEN
        PopRegisterStack;
    FI;

```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.

$\pm\infty / \pm\infty$; $\pm 0 / \pm 0$

#D Source is a denormal value.

#Z	DEST / ± 0 , where DEST is not equal to ± 0 .
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /7	FDIVR <i>m32fp</i>	Valid	Valid	Divide <i>m32fp</i> by ST(0) and store result in ST(0).
DC /7	FDIVR <i>m64fp</i>	Valid	Valid	Divide <i>m64fp</i> by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(0), ST(i)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(0).
DC F0+i	FDIVR ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(i).
DE F0+i	FDIVRP ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i), store result in ST(i), and pop the register stack.
DE F1	FDIVRP	Valid	Valid	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack.
DA /7	FIDIVR <i>m32int</i>	Valid	Valid	Divide <i>m32int</i> by ST(0) and store result in ST(0).
DE /7	FIDIVR <i>m16int</i>	Valid	Valid	Divide <i>m16int</i> by ST(0) and store result in ST(0).

Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-34. FDIVR/FDIVRP/FIDIVR Results

SRC		DEST							
			$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
		$-\infty$	*	$+\infty$	$+\infty$	$-\infty$	$-\infty$	*	NaN
		$-F$	$+0$	$+F$	**	**	$-F$	-0	NaN
		$-I$	$+0$	$+F$	**	**	$-F$	-0	NaN
		-0	$+0$	$+0$	*	*	-0	-0	NaN
		$+0$	-0	-0	*	*	$+0$	$+0$	NaN
		$+I$	-0	$-F$	**	**	$+F$	$+0$	NaN
		$+F$	-0	$-F$	**	**	$+F$	$+0$	NaN
		$+\infty$	*	$-\infty$	$-\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF DEST = 0
    THEN
        #Z;
    ELSE
        IF Instruction = FIDIVR
            THEN
                DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
            ELSE (* Source operand is floating-point value *)
                DEST ← SRC / DEST;
        FI;
    FI;

```

```

IF Instruction = FDIVRP
    THEN
        PopRegisterStack;
    FI;

```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
 $\pm\infty / \pm\infty$; $\pm 0 / \pm 0$

#D	Source is a denormal value.
#Z	SRC / ± 0 , where SRC is not equal to ± 0 .
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FFREE—Free Floating-Point Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD C0+i	FFREE ST(i)	Valid	Valid	Sets tag for ST(i) to empty.

Description

Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

TAG(i) ← 11B;

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FICOM/FICOMP—Compare Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DE /2	FICOM <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> .
DA /2	FICOM <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> .
DE /3	FICOMP <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> and pop stack register.
DA /3	FICOMP <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> and pop stack register.

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

Table 3-35. FICOM/FICOMP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to “unordered.”

The sign of zero is ignored, so that $-0.0 \leftarrow +0.0$.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

CASE (relation of operands) OF

ST(0) > SRC: C3, C2, C0 \leftarrow 000;

ST(0) < SRC: C3, C2, C0 \leftarrow 001;

ST(0) = SRC: C3, C2, C0 \leftarrow 100;

Unordered: C3, C2, C0 \leftarrow 111;

ESAC;

IF Instruction = FICOMP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 See table on previous page.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA One or both operands are NaN values or have unsupported formats.

#D One or both operands are denormal values.

Protected Mode Exceptions

#GP(0) 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.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#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.

#UD If the LOCK prefix is used.

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.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#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.

#UD If the LOCK prefix is used.

FILD—Load Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /0	FILD <i>m16int</i>	Valid	Valid	Push <i>m16int</i> onto the FPU register stack.
DB /0	FILD <i>m32int</i>	Valid	Valid	Push <i>m32int</i> onto the FPU register stack.
DF /5	FILD <i>m64int</i>	Valid	Valid	Push <i>m64int</i> onto the FPU register stack.

Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

TOP \leftarrow TOP – 1;
 ST(0) \leftarrow ConvertToDoubleExtendedPrecisionFP(SRC);

FPU Flags Affected

C1 Set to 1 if stack overflow occurred; set to 0 otherwise.
 C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack overflow occurred.

Protected Mode Exceptions

#GP(0) 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.
 #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #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.
 #UD If the LOCK prefix is used.

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.
 #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
 #SS(0) If a memory operand effective address is outside the SS segment limit.
 #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #PF(fault-code) If a page fault occurs.
 #AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #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.
- #UD If the LOCK prefix is used.

FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F7	FINCSTP	Valid	Valid	Increment the TOP field in the FPU status register.

Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF TOP = 7
    THEN TOP ← 0;
    ELSE TOP ← TOP + 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #MF If there is a pending x87 FPU exception.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FINIT/FNINIT—Initialize Floating-Point Unit

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DB E3	FINIT	Valid	Valid	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT*	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

Operation

```

FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;

```

FPU Flags Affected

C0, C1, C2, C3 set to 0.

Floating-Point Exceptions

None.

Protected Mode Exceptions

```

#NM      CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF      If there is a pending x87 FPU exception.

```

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FIST/FISTP—Store Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /2	FIST <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> .
DB /2	FIST <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> .
DF /3	FISTP <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> and pop register stack.
DB /3	FISTP <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> and pop register stack.
DF /7	FISTP <i>m64int</i>	Valid	Valid	Store ST(0) in <i>m64int</i> and pop register stack.

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-36. FIST/FISTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < -0$	**
-0	0
$+0$	0
$+0 < F < +1$	**
$F \geq +1$	$+I$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:
 F Means finite floating-point value.
 I Means integer.
 * Indicates floating-point invalid-operation (#IA) exception.
 ** 0 or ± 1 , depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞ , SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST \leftarrow Integer(ST(0));

IF Instruction = FISTP
 THEN
 PopRegisterStack;
 FI;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the inexact exception (#P) is generated: 0 \leftarrow not roundup; 1 \leftarrow roundup. Set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format. Source operand is an SNaN, QNaN, $\pm\infty$, or unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FISTTP—Store Integer with Truncation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /1	FISTTP <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> with truncation.
DB /1	FISTTP <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> with truncation.
DD /1	FISTTP <i>m64int</i>	Valid	Valid	Store ST(0) in <i>m64int</i> with truncation.

Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-37. FISTTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < +1$	0
$F \geq +1$	$+I$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST \leftarrow ST;

pop ST;

Flags Affected

C1 is cleared; C0, C2, C3 undefined.

Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

Protected Mode Exceptions

#GP(0)	If the destination is in a nonwritable segment.
	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
 If the LOCK prefix is used.

Real Address Mode Exceptions

GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
 #NM If CR0.EM[bit 2] = 1.
 If CR0.TS[bit 3] = 1.
 #UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
 If the LOCK prefix is used.

Virtual 8086 Mode Exceptions

GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
 #NM If CR0.EM[bit 2] = 1.
 If CR0.TS[bit 3] = 1.
 #UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
 If the LOCK prefix is used.
 #PF(fault-code) For a page fault.
 #AC(0) For unaligned memory reference if the current privilege is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
 #GP(0) If the memory address is in a non-canonical form.
 #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #MF If there is a pending x87 FPU exception.
 #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.
 If the LOCK prefix is used.

FLD—Load Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /0	FLD <i>m32fp</i>	Valid	Valid	Push <i>m32fp</i> onto the FPU register stack.
DD /0	FLD <i>m64fp</i>	Valid	Valid	Push <i>m64fp</i> onto the FPU register stack.
DB /5	FLD <i>m80fp</i>	Valid	Valid	Push <i>m80fp</i> onto the FPU register stack.
D9 C0+i	FLD ST(i)	Valid	Valid	Push ST(i) onto the FPU register stack.

Description

Pushes the source operand onto the FPU register stack. The source operand can be in single-precision, double-precision, or double extended-precision floating-point format. If the source operand is in single-precision or double-precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF SRC is ST(i)
    THEN
        temp ← ST(i);
FI;

TOP ← TOP – 1;

IF SRC is memory-operand
    THEN
        ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) ← temp;
FI;
```

FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow or overflow occurred.
#IA Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD *m80fp* or FLD ST(i)).
#D Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

Protected Mode Exceptions

#GP(0) If destination is located in a non-writable 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.

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E8	FLD1	Valid	Valid	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Valid	Valid	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Valid	Valid	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Valid	Valid	Push π onto the FPU register stack.
D9 EC	FLDLG2	Valid	Valid	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Valid	Valid	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Valid	Valid	Push +0.0 onto the FPU register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, π , $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled “Approximation of Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the π constant.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

Operation

TOP \leftarrow TOP – 1;
ST(0) \leftarrow CONSTANT;

FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack overflow occurred.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FLDCW—Load x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /5	FLDCW <i>m2byte</i>	Valid	Valid	Load FPU control word from <i>m2byte</i> .

Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmask one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

FPUControlWord ← SRC;

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FLDENV—Load x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /4	FLDENV <i>m14/28byte</i>	Valid	Valid	Load FPU environment from <i>m14byte</i> or <i>m28byte</i> .

Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next “waiting” floating-point instruction.

Protected Mode Exceptions

#GP(0)	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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /1	FMUL <i>m32fp</i>	Valid	Valid	Multiply ST(0) by <i>m32fp</i> and store result in ST(0).
DC /1	FMUL <i>m64fp</i>	Valid	Valid	Multiply ST(0) by <i>m64fp</i> and store result in ST(0).
D8 C8+i	FMUL ST(0), ST(i)	Valid	Valid	Multiply ST(0) by ST(i) and store result in ST(0).
DC C8+i	FMUL ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0) and store result in ST(i).
DE C8+i	FMULP ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE C9	FMULP	Valid	Valid	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /1	FIMUL <i>m32int</i>	Valid	Valid	Multiply ST(0) by <i>m32int</i> and store result in ST(0).
DE /1	FIMUL <i>m16int</i>	Valid	Valid	Multiply ST(0) by <i>m16int</i> and store result in ST(0).

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞ . When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-38. FMUL/FMULP/FIMUL Results

SRC	DEST						
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$-\infty$	$+\infty$	$+\infty$	*	*	$-\infty$	$-\infty$
	$-F$	$+\infty$	$+F$	$+0$	-0	$-F$	$-\infty$
	$-I$	$+\infty$	$+F$	$+0$	-0	$-F$	$-\infty$
	-0	*	$+0$	$+0$	-0	-0	*
	$+0$	*	-0	-0	$+0$	$+0$	*
	$+I$	$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$+F$	$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$+\infty$	$-\infty$	$-\infty$	*	*	$+\infty$	$+\infty$
	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means Integer.

* Indicates invalid-arithmetc-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF Instruction = FIMUL

THEN

DEST \leftarrow DEST * ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST \leftarrow DEST * SRC;

FI;

IF Instruction = FMULP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.
One operand is ± 0 and the other is $\pm\infty$.
- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FNOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 D0	FNOP	Valid	Valid	No operation is performed.

Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register and the FPU Instruction Pointer.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPATAN—Partial Arctangent

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F3	FPATAN	Valid	Valid	Replace ST(1) with $\arctan(\text{ST}(1)/\text{ST}(0))$ and pop the register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than $+\pi$.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point $(-X,Y)$ is in the second quadrant, resulting in an angle between $\pi/2$ and π , while a point $(X,-Y)$ is in the fourth quadrant, resulting in an angle between 0 and $-\pi/2$. A point $(-X,-Y)$ is in the third quadrant, giving an angle between $-\pi/2$ and $-\pi$.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 3-39. FPATAN Results

		ST(0)						
ST(1)		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	$-3\pi/4^*$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4^*$	NaN
	$-F$	$-p$	$-\pi$ to $-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$ to -0	-0	NaN
	-0	$-p$	$-p$	$-p^*$	-0^*	-0	-0	NaN
	$+0$	$+p$	$+p$	$+\pi^*$	$+0^*$	$+0$	$+0$	NaN
	$+F$	$+p$	$+\pi$ to $+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$ to $+0$	$+0$	NaN
	$+\infty$	$+3\pi/4^*$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4^*$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Table 8-10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, specifies that the ratios $0/0$ and ∞/∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the $0/0$ or ∞/∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, $\arctan(0,0)$ etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$$0 \leq |\text{ST}(1)| < |\text{ST}(0)| < +\infty$$

Operation

```
ST(1) ← arctan(ST(1) / ST(0));  
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPREM—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F8	FPREM	Valid	Valid	Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1).

Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} \leftarrow \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of $[\text{ST}(0) / \text{ST}(1)]$ toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-40. FPREM Results

		ST(1)						
ST(0)		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	$-\infty$	*	*	*	*	*	*	NaN
	-F	ST(0)	-F or -0	**	**	-F or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞ , the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$D \leftarrow \text{exponent}(\text{ST}(0)) - \text{exponent}(\text{ST}(1));$

IF $D < 64$

THEN

$Q \leftarrow \text{Integer}(\text{TruncateTowardZero}(\text{ST}(0) / \text{ST}(1)));$

$\text{ST}(0) \leftarrow \text{ST}(0) - (\text{ST}(1) * Q);$

$C2 \leftarrow 0;$

$C0, C3, C1 \leftarrow \text{LeastSignificantBits}(Q); (* Q2, Q1, Q0 *)$

ELSE

$C2 \leftarrow 1;$

$N \leftarrow \text{An implementation-dependent number between 32 and 63};$

$QQ \leftarrow \text{Integer}(\text{TruncateTowardZero}((\text{ST}(0) / \text{ST}(1)) / 2^{(D-N)}));$

$\text{ST}(0) \leftarrow \text{ST}(0) - (\text{ST}(1) * QQ * 2^{(D-N)});$

FI;

FPU Flags Affected

C0 Set to bit 2 (Q2) of the quotient.

C1 Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).

C2 Set to 0 if reduction complete; set to 1 if incomplete.

C3 Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value, modulus is 0, dividend is ∞ , or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPREM1—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F5	FPREM1	Valid	Valid	Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1).

Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} \leftarrow \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of $[\text{ST}(0) / \text{ST}(1)]$ toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-41. FPREM1 Results

		ST(1)						
ST(0)		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	*	*	*	*	*	NaN
	$-F$	ST(0)	$\pm F$ or -0	**	**	$\pm F$ or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	$+0$	$+0$	$+0$	*	*	$+0$	$+0$	NaN
	$+F$	ST(0)	$\pm F$ or $+0$	**	**	$\pm F$ or $+0$	ST(0)	NaN
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞ , the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the “Operation” section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$D \leftarrow \text{exponent}(\text{ST}(0)) - \text{exponent}(\text{ST}(1));$

IF $D < 64$

THEN

$Q \leftarrow \text{Integer}(\text{RoundTowardNearestInteger}(\text{ST}(0) / \text{ST}(1)));$

$\text{ST}(0) \leftarrow \text{ST}(0) - (\text{ST}(1) * Q);$

$C2 \leftarrow 0;$

$C0, C3, C1 \leftarrow \text{LeastSignificantBits}(Q); (* Q2, Q1, Q0 *)$

ELSE

$C2 \leftarrow 1;$

$N \leftarrow \text{An implementation-dependent number between 32 and 63};$

$QQ \leftarrow \text{Integer}(\text{TruncateTowardZero}((\text{ST}(0) / \text{ST}(1)) / 2^{(D-N)}));$

$\text{ST}(0) \leftarrow \text{ST}(0) - (\text{ST}(1) * QQ * 2^{(D-N)});$

FI;

FPU Flags Affected

C0 Set to bit 2 (Q2) of the quotient.

C1 Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).

C2 Set to 0 if reduction complete; set to 1 if incomplete.

C3 Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞ , or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPTAN—Partial Tangent

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F2	FPTAN	Valid	Valid	Replace ST(0) with its approximate tangent and push 1 onto the FPU stack.

Description

Computes the approximate tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

Table 3-42. FPTAN Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
$-F$	$-F$ to $+F$
-0	-0
$+0$	$+0$
$+F$	$-F$ to $+F$
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF ST(0) < 263
  THEN
    C2 ← 0;
    ST(0) ← fptan(ST(0)); // approximation of tan
    TOP ← TOP - 1;
    ST(0) ← 1.0;
  ELSE (* Source operand is out-of-range *)
    C2 ← 1;

```

FI;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FRNDINT—Round to Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FC	FRNDINT	Valid	Valid	Round ST(0) to an integer.

Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞ , the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$ST(0) \leftarrow \text{RoundToIntegerValue}(ST(0));$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FRSTOR—Restore x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD /4	FRSTOR <i>m94/108byte</i>	Valid	Valid	Load FPU state from <i>m94byte</i> or <i>m108byte</i> .

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

```

```

ST(0) ← SRC[ST(0)];
ST(1) ← SRC[ST(1)];
ST(2) ← SRC[ST(2)];
ST(3) ← SRC[ST(3)];
ST(4) ← SRC[ST(4)];
ST(5) ← SRC[ST(5)];
ST(6) ← SRC[ST(6)];
ST(7) ← SRC[ST(7)];

```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

Protected Mode Exceptions

#GP(0) 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

FSAVE/FNSAVE—Store x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DD /6	FSAVE <i>m94/108byte</i>	Valid	Valid	Store FPU state to <i>m94byte</i> or <i>m108byte</i> after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE* <i>m94/108byte</i>	Valid	Valid	Store FPU environment to <i>m94byte</i> or <i>m108byte</i> without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see “FINIT/FNINIT—Initialize Floating-Point Unit” in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a “clean” FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

For Intel math coprocessors and FPU prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

(* Save FPU State and Registers *)

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUInstructionPointer] ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;
```

```
DEST[ST(0)] ← ST(0);
DEST[ST(1)] ← ST(1);
DEST[ST(2)] ← ST(2);
DEST[ST(3)] ← ST(3);
DEST[ST(4)] ← ST(4);
DEST[ST(5)] ← ST(5);
DEST[ST(6)] ← ST(6);
DEST[ST(7)] ← ST(7);
```

(* Initialize FPU *)

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	<p>If destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.

FSCALE—Scale

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FD	FSCALE	Valid	Valid	Scale ST(0) by ST(1).

Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-43. FSCALE Results

ST(0)	ST(1)							
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	NaN	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	-0	$-F$	$-F$	$-F$	$-F$	$-\infty$	NaN
	-0	-0	-0	-0	-0	-0	NaN	NaN
	$+0$	$+0$	$+0$	$+0$	$+0$	$+0$	NaN	NaN
	$+F$	$+0$	$+F$	$+F$	$+F$	$+F$	$+\infty$	NaN
	$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$$ST(0) \leftarrow ST(0) * 2^{\text{RoundTowardZero}(ST(1))};$$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSIN—Sine

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FE	FSIN	Valid	Valid	Replace ST(0) with the approximate of its sine.

Description

Computes an approximation of the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

Table 3-44. FSIN Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	-1 to $+1$
-0	-0
$+0$	$+0$
$+F$	-1 to $+1$
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/4$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF  $-2^{63} < \text{ST}(0) < 2^{63}$ 
  THEN
    C2  $\leftarrow$  0;
    ST(0)  $\leftarrow$  fsin(ST(0)); // approximation of the mathematical sin function
  ELSE (* Source operand out of range *)
    C2  $\leftarrow$  1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSINCOS—Sine and Cosine

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FB	FSINCOS	Valid	Valid	Compute the sine and cosine of ST(0); replace ST(0) with the approximate sine, and push the approximate cosine onto the register stack.

Description

Computes both the approximate sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

Table 3-45. FSINCOS Results

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
$-F$	-1 to $+1$	-1 to $+1$
-0	$+1$	-0
$+0$	$+1$	$+0$
$+F$	-1 to $+1$	-1 to $+1$
$+\infty$	*	*
NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF ST(0) <  $2^{63}$ 
  THEN
    C2 ← 0;
    TEMP ← fcos(ST(0)); // approximation of cosine
    ST(0) ← fsin(ST(0)); // approximation of sine
    TOP ← TOP – 1;
    ST(0) ← TEMP;
  ELSE (* Source operand out of range *)

```

$C2 \leftarrow 1;$
 FI;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurs. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSQRT—Square Root

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FA	FSQRT	Valid	Valid	Computes square root of ST(0) and stores the result in ST(0).

Description

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-46. FSQRT Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	*
-0	-0
$+0$	$+0$
$+F$	$+F$
$+\infty$	$+\infty$
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$ST(0) \leftarrow \text{SquareRoot}(ST(0));$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
	Source operand is a negative value (except for -0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FST/FSTP—Store Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /2	FST <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> .
DD /2	FST <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> .
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> and pop register stack.
DD /3	FSTP <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> and pop register stack.
DB /7	FSTP <i>m80fp</i>	Valid	Valid	Copy ST(0) to <i>m80fp</i> and pop register stack.
DD D8+i	FSTP ST(i)	Valid	Valid	Copy ST(0) to ST(i) and pop register stack.

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single-precision or double-precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single-precision or double-precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ± 0 , $\pm\infty$, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0, ∞ , or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST \leftarrow ST(0);

```
IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 \leftarrow not roundup; 1 \leftarrow roundup.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B D9 /7	FSTCW <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

DEST ← FPUControlWord;

FPU Flags Affected

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

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FSTENV/FNSTENV—Store x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B D9 /6	FSTENV <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV* <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUInstructionPointer] ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;
```

FPU Flags Affected

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

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DD /7	FSTSW <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Valid	Valid	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW* <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled “Branching and Conditional Moves on FPU Condition Codes” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

DEST ← FPUStatusWord;

FPU Flags Affected

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

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /4	FSUB <i>m32fp</i>	Valid	Valid	Subtract <i>m32fp</i> from ST(0) and store result in ST(0).
DC /4	FSUB <i>m64fp</i>	Valid	Valid	Subtract <i>m64fp</i> from ST(0) and store result in ST(0).
D8 E0+i	FSUB ST(0), ST(i)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(0).
DC E8+i	FSUB ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(i).
DE E8+i	FSUBP ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.
DE E9	FSUBP	Valid	Valid	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.
DA /4	FISUB <i>m32int</i>	Valid	Valid	Subtract <i>m32int</i> from ST(0) and store result in ST(0).
DE /4	FISUB <i>m16int</i>	Valid	Valid	Subtract <i>m16int</i> from ST(0) and store result in ST(0).

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-47 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is -0 . This instruction also guarantees that $+0 - (-0) = +0$, and that $-0 - (+0) = -0$. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞ , the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-47. FSUB/FSUBP/FISUB Results

DEST	SRC							
		$-\infty$	$-F$ or $-I$	-0	$+0$	$+F$ or $+I$	$+\infty$	NaN
	$-\infty$	*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	$+\infty$	$\pm F$ or ± 0	DEST	DEST	$-F$	$-\infty$	NaN
	-0	$+\infty$	$-\text{SRC}$	± 0	-0	$-\text{SRC}$	$-\infty$	NaN
	$+0$	$+\infty$	$-\text{SRC}$	$+0$	± 0	$-\text{SRC}$	$-\infty$	NaN
	$+F$	$+\infty$	$+F$	DEST	DEST	$\pm F$ or ± 0	$-\infty$	NaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF Instruction = FISUB

THEN

DEST \leftarrow DEST – ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST \leftarrow DEST – SRC;

FI;

IF Instruction = FSUBP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.
Operands are infinities of like sign.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /5	FSUBR <i>m32fp</i>	Valid	Valid	Subtract ST(0) from <i>m32fp</i> and store result in ST(0).
DC /5	FSUBR <i>m64fp</i>	Valid	Valid	Subtract ST(0) from <i>m64fp</i> and store result in ST(0).
D8 E8+i	FSUBR ST(0), ST(i)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(0).
DC E0+i	FSUBR ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(i).
DE E0+i	FSUBRP ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack.
DE E1	FSUBRP	Valid	Valid	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack.
DA /5	FISUBR <i>m32int</i>	Valid	Valid	Subtract ST(0) from <i>m32int</i> and store result in ST(0).
DE /5	FISUBR <i>m16int</i>	Valid	Valid	Subtract ST(0) from <i>m16int</i> and store result in ST(0).

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is -0 . This instruction also guarantees that $+0 - (-0) = +0$, and that $-0 - (+0) = -0$. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞ , the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-48. FSUBR/FSUBRP/FISUBR Results

DEST	SRC							
		$-\infty$	$-F$ or $-I$	-0	$+0$	$+F$ or $+I$	$+\infty$	NaN
	$-\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	$-F$	$-\infty$	$\pm F$ or ± 0	$-\text{DEST}$	$-\text{DEST}$	$+F$	$+\infty$	NaN
	-0	$-\infty$	SRC	± 0	$+0$	SRC	$+\infty$	NaN
	$+0$	$-\infty$	SRC	-0	± 0	SRC	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	$-\text{DEST}$	$-\text{DEST}$	$\pm F$ or ± 0	$+\infty$	NaN
	$+\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF Instruction = FISUBR

THEN

DEST \leftarrow ConvertToDoubleExtendedPrecisionFP(SRC) $-$ DEST;

ELSE (* Source operand is floating-point value *)

DEST \leftarrow SRC $-$ DEST; FI;

IF Instruction = FSUBRP

THEN

PopRegisterStack; FI;

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.
Operands are infinities of like sign.
- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

- #GP(0) 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.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#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.
#UD	If the LOCK prefix is used.

FTST—TEST

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E4	FTST	Valid	Valid	Compare ST(0) with 0.0.

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

Table 3-49. FTST Results

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered” and the invalid operation exception is generated.

The sign of zero is ignored, so that $(-0.0 \leftarrow +0.0)$.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

CASE (relation of operands) OF

Not comparable: C3, C2, C0 \leftarrow 111;

ST(0) > 0.0: C3, C2, C0 \leftarrow 000;

ST(0) < 0.0: C3, C2, C0 \leftarrow 001;

ST(0) = 0.0: C3, C2, C0 \leftarrow 100;

ESAC;

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 See Table 3-49.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA The source operand is a NaN value or is in an unsupported format.

#D The source operand is a denormal value.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD E0+i	FUCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
DD E1	FUCOM	Valid	Valid	Compare ST(0) with ST(1).
DD E8+i	FUCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
DD E9	FUCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DA E9	FUCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that -0.0 is equal to $+0.0$.

Table 3-50. FUCOM/FUCOMP/FUCOMPP Results

Comparison Results*	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

CASE (relation of operands) OF

ST > SRC: C3, C2, C0 \leftarrow 000;

ST < SRC: C3, C2, C0 \leftarrow 001;

ST = SRC: C3, C2, C0 \leftarrow 100;

ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format

THEN

C3, C2, C0 \leftarrow 111;

ELSE (* ST(0) or SRC is SNaN or unsupported format *)

#IA;

```
IF FPUControlWord.IM = 1
    THEN
        C3, C2, C0 ← 111;
FI;

FI;

IF Instruction = FUCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FUCOMPP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
- C0, C2, C3 See Table 3-50.

Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
- #D One or both operands are denormal values.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXAM—Examine ModR/M

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E5	FXAM	Valid	Valid	Classify value or number in ST(0).

Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

Table 3-51. FXAM Results

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

C1 ← sign bit of ST; (* 0 for positive, 1 for negative *)

CASE (class of value or number in ST(0)) OF

 Unsupported: C3, C2, C0 ← 000;

 NaN: C3, C2, C0 ← 001;

 Normal: C3, C2, C0 ← 010;

 Infinity: C3, C2, C0 ← 011;

 Zero: C3, C2, C0 ← 100;

 Empty: C3, C2, C0 ← 101;

 Denormal: C3, C2, C0 ← 110;

ESAC;

FPU Flags Affected

C1 Sign of value in ST(0).

C0, C2, C3 See Table 3-51.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXCH—Exchange Register Contents

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 C8+i	FXCH ST(i)	Valid	Valid	Exchange the contents of ST(0) and ST(i).
D9 C9	FXCH	Valid	Valid	Exchange the contents of ST(0) and ST(1).

Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF (Number-of-operands) is 1

THEN

```
temp ← ST(0);
ST(0) ← SRC;
SRC ← temp;
```

ELSE

```
temp ← ST(0);
ST(0) ← ST(1);
ST(1) ← temp;
```

FI;

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXRSTOR—Restore x87 FPU, MMX, XMM, and MXCSR State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /1 FXRSTOR <i>m512byte</i>	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .
REX.W+ OF AE /1 FXRSTOR64 <i>m512byte</i>	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-52 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-55 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-56 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-52, Table 3-55, or Table 3-56. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

Operation

IF 64-Bit Mode

THEN

(x87 FPU, MMX, XMM15-XMM0, MXCSR) Load(SRC);

ELSE

(x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);

FI;

x87 FPU and SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See alignment check exception [#AC] below.) For an attempt to set reserved bits in MXCSR.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH. For an attempt to set reserved bits in MXCSR.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment. For an attempt to set reserved bits in MXCSR.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.

#AC

If instruction is preceded by a LOCK prefix.

If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

FXSAVE—Save x87 FPU, MMX Technology, and SSE State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /0 FXSAVE <i>m512byte</i>	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .
REX.W+ OF AE /0 FXSAVE64 <i>m512byte</i>	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area. The operation of FXSAVE in non-64-bit modes is described first.

Non-64-Bit Mode Operation

Table 3-52 shows the layout of the state information in memory when the processor is operating in legacy modes.

**Table 3-52. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR
Memory Region**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsvd		FPU CS		FPU IP				FOP		Rsvd	FTW	FSW		FCW		0
MXCSR_MASK				MXCSR				Rsvd		FPU DS		FPU DP				16
Reserved						ST0/MM0										32
Reserved						ST1/MM1										48
Reserved						ST2/MM2										64
Reserved						ST3/MM3										80
Reserved						ST4/MM4										96
Reserved						ST5/MM5										112
Reserved						ST6/MM6										128
Reserved						ST7/MM7										144
XMM0																160
XMM1																176
XMM2																192
XMM3																208
XMM4																224
XMM5																240
XMM6																256
XMM7																272

**Table 3-52. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR
Memory Region (Contd.)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved																288
Reserved																304
Reserved																320
Reserved																336
Reserved																352
Reserved																368
Reserved																384
Reserved																400
Reserved																416
Reserved																432
Reserved																448
Available																464
Available																480
Available																496

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-52 are defined in Table 3-53.

Table 3-53. Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU opcode field.
FPU IP	x87 FPU Instruction Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for a description of the x87 FPU instruction pointer.
FPU CS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates the FPU CS and FPU DS values, and this field is saved as 0000H.

Table 3-53. Field Definitions (Contd.)

Field	Definition
FPU DP	x87 FPU Instruction Operand (Data) Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for a description of the x87 FPU operand pointer.
FPU DS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates the FPU CS and FPU DS values, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See “Guidelines for Writing to the MXCSR Register” in Chapter 11 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

```

R7 R6 R5 R4 R3 R2 R1 R0
11 xx xx xx 11 11 11 11

```

Here, 11B indicates empty stack elements and “xx” indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

```

R7 R6 R5 R4 R3 R2 R1 R0
0  1  1  1  0  0  0  0

```

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be used by an application program to pass a “clean” x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute a FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.
- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management).

This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-54.

Table 3-54. Recreating FSAVE Format

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW
0	0	0	0x	1	Special 10
0	0	0	1x	1	Valid 00
0	0	1	00	1	Special 10
0	0	1	10	1	Valid 00
0	1	0	0x	1	Special 10
0	1	0	1x	1	Special 10
0	1	1	00	1	Zero 01
0	1	1	10	1	Special 10
1	0	0	1x	1	Special 10
1	0	0	1x	1	Special 10
1	0	1	00	1	Special 10
1	0	1	10	1	Special 10
For all legal combinations above.				0	Empty 11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-55), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-56), the FPU IP and FPU DP pointers are 32-bits.

**Table 3-55. Layout of the 64-bit-mode FXSAVE64 Map
(requires REX.W = 1)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
FPU IP								FOP	Reserved	FTW	FSW	FCW	0				
MXCSR_MASK				MXCSR				FPU DP								16	
Reserved						ST0/MM0										32	
Reserved						ST1/MM1										48	
Reserved						ST2/MM2										64	
Reserved						ST3/MM3										80	
Reserved						ST4/MM4										96	
Reserved						ST5/MM5										112	

**Table 3-55. Layout of the 64-bit-mode FXSAVE64 Map
(requires REX.W = 1) (Contd.)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved						ST6/MM6										128
Reserved						ST7/MM7										144
XMM0																160
XMM1																176
XMM2																192
XMM3																208
XMM4																224
XMM5																240
XMM6																256
XMM7																272
XMM8																288
XMM9																304
XMM10																320
XMM11																336
XMM12																352
XMM13																368
XMM14																384
XMM15																400
Reserved																416
Reserved																432
Reserved																448
Available																464
Available																480
Available																496

Table 3-56. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Reserved		FPU CS		FPU IP				FOP		Reserved	FTW		FSW		FCW		0
MXCSR_MASK				MXCSR				Reserved		FPU DS			FPU DP				16
Reserved						ST0/MM0										32	
Reserved						ST1/MM1										48	
Reserved						ST2/MM2										64	
Reserved						ST3/MM3										80	
Reserved						ST4/MM4										96	
Reserved						ST5/MM5										112	
Reserved						ST6/MM6										128	
Reserved						ST7/MM7										144	
XMM0																160	

Table 3-56. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0) (Contd.) (Contd.)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
XMM1																176
XMM2																192
XMM3																208
XMM4																224
XMM5																240
XMM6																256
XMM7																272
XMM8																288
XMM9																304
XMM10																320
XMM11																336
XMM12																352
XMM13																368
XMM14																384
XMM15																400
Reserved																416
Reserved																432
Reserved																448
Available																464
Available																480
Available																496

Operation

```

IF 64-Bit Mode
  THEN
    IF REX.W = 1
      THEN
        DEST ← Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0,
        MXCSR);
      ELSE
        DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
    FI;
  ELSE
    DEST ← SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
  FI;

```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the description of the alignment check exception [#AC] below.)
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.

	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
#UD	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Implementation Note

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This order vary for FXSAVE for different processor implementations.

FXTRACT—Extract Exponent and Significand

Opcode/ Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F4 FXTRACT	Valid	Valid	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended $\log_b(x)$ function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of $-\infty$ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP ← TOP – 1;
ST(0) ← TEMP;
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow or overflow occurred.
#IA Source operand is an SNaN value or unsupported format.
#Z ST(0) operand is ± 0 .
#D Source operand is a denormal value.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FYL2X—Compute $y * \log_2 x$

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F1	FYL2X	Valid	Valid	Replace ST(1) with $(ST(1) * \log_2 ST(0))$ and pop the register stack.

Description

Computes $(ST(1) * \log_2 (ST(0)))$, stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-57. FYL2X Results

		ST(0)							
ST(1)		$-\infty$	$-F$	± 0	$+0 < +F < +1$	$+1$	$+F > +1$	$+\infty$	NaN
	$-\infty$	*	*	$+\infty$	$+\infty$	*	$-\infty$	$-\infty$	NaN
	$-F$	*	*	**	$+F$	-0	$-F$	$-\infty$	NaN
	-0	*	*	*	$+0$	-0	-0	*	NaN
	$+0$	*	*	*	-0	$+0$	$+0$	*	NaN
	$+F$	*	*	**	$-F$	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	*	$-\infty$	$-\infty$	*	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ± 0 , the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$ST(1) \leftarrow ST(1) * \log_2 ST(0);$

PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
-----	---------------------------

#IA	Either operand is an SNaN or unsupported format. Source operand in register ST(0) is a negative finite value (not -0).
#Z	Source operand in register ST(0) is ± 0 .
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FYL2XP1—Compute $y * \log_2(x + 1)$

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F9	FYL2XP1	Valid	Valid	Replace ST(1) with $ST(1) * \log_2(ST(0) + 1.0)$ and pop the register stack.

Description

Computes $(ST(1) * \log_2(ST(0) + 1.0))$, stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1 - \sqrt{2}/2) \text{ to } (1 - \sqrt{2}/2)$$

The source operand in ST(1) can range from $-\infty$ to $+\infty$. If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

Table 3-58. FYL2XP1 Results

		ST(0)				
ST(1)		$-(1 - (\sqrt{2}/2)) \text{ to } -0$	-0	+0	+0 to $+(1 - (\sqrt{2}/2))$	NaN
	$-\infty$	$+\infty$	*	*	$-\infty$	NaN
	-F	+F	+0	-0	-F	NaN
	-0	+0	+0	-0	-0	NaN
	+0	-0	-0	+0	+0	NaN
	+F	-F	-0	+0	+F	NaN
	$+\infty$	$-\infty$	*	*	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (ϵ) values, more significant digits can be retained by using the FYL2XP1 instruction than by using $(\epsilon+1)$ as an argument to the FYL2X instruction. The $(\epsilon+1)$ expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

$$\text{scale factor} \leftarrow \log_n 2$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

$ST(1) \leftarrow ST(1) * \log_2(ST(0) + 1.0);$

PopRegisterStack;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HADDPD—Packed Double-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE3	Horizontal add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7C /r VHADDPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7C /r VHADDPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

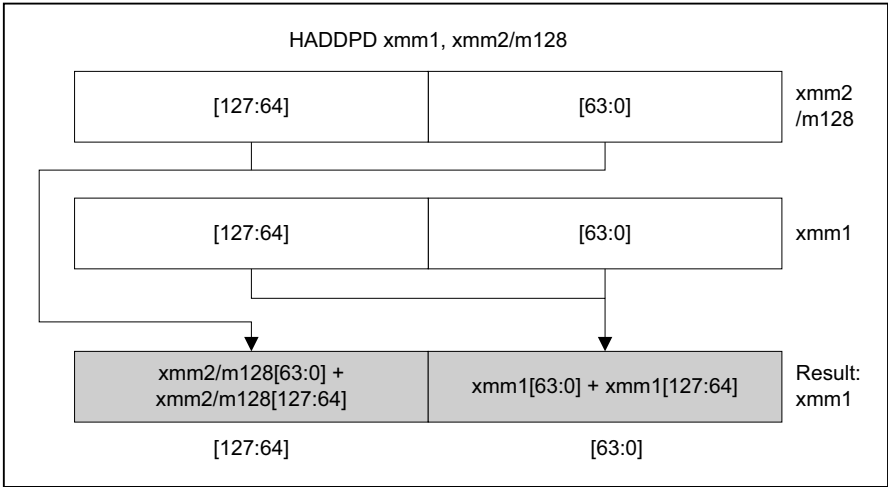
Description

Adds the double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-15 for HADDPD; see Figure 3-16 for VHADDPD.



OM15993

Figure 3-15. HADDPD—Packed Double-FP Horizontal Add

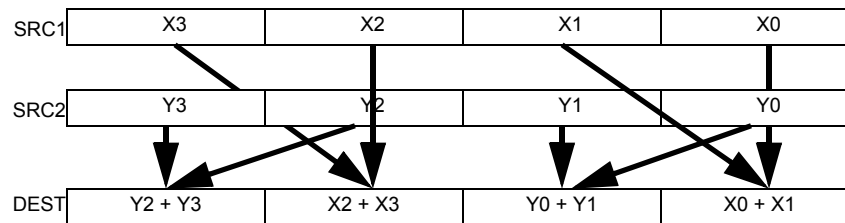


Figure 3-16. VHADDPD operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HADDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[VLMAX-1:128] (Unmodified)
```

VHADDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[VLMAX-1:128] ← 0
```

VHADDPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[191:128] ← SRC1[255:192] + SRC1[191:128]
DEST[255:192] ← SRC2[255:192] + SRC2[191:128]
```

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD: `__m256d _mm256_hadd_pd (__m256d a, __m256d b);`

HADDPD: `__m128d _mm_hadd_pd (__m128d a, __m128d b);`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

HADDPS—Packed Single-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7C /r HADDPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE3	Horizontal add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7C /r VHADDPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Horizontal add packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7C /r VHADDPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Horizontal add packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-17 for HADDPS; see Figure 3-18 for VHADDPS.

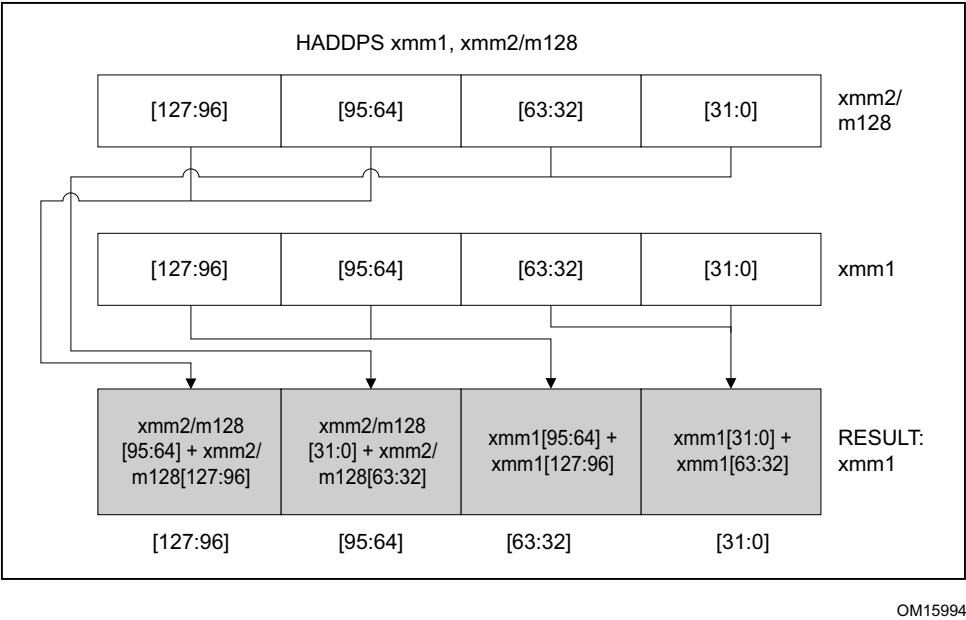


Figure 3-17. HADDPS—Packed Single-FP Horizontal Add

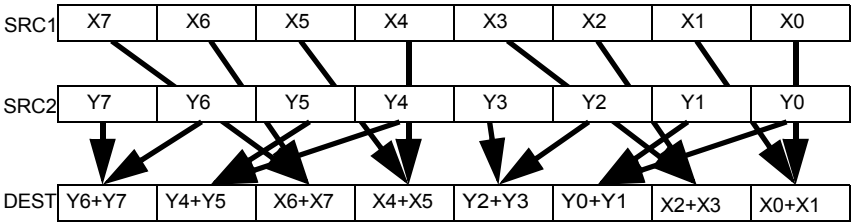


Figure 3-18. VHADDPS operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HADDPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VHADDPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VHADDPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[191:160] + \text{SRC1}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[255:224] + \text{SRC1}[223:192]$
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[191:160] + \text{SRC2}[159:128]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[255:224] + \text{SRC2}[223:192]$

Intel C/C++ Compiler Intrinsic Equivalent

HADDPS: `__m128 _mm_hadd_ps (__m128 a, __m128 b);`

VHADDPS: `__m256 _mm256_hadd_ps (__m256 a, __m256 b);`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

HLT—Halt

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F4	HLT	NP	Valid	Valid	Halt

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

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

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

Enter Halt state;

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HSUBPD—Packed Double-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Horizontal subtract packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

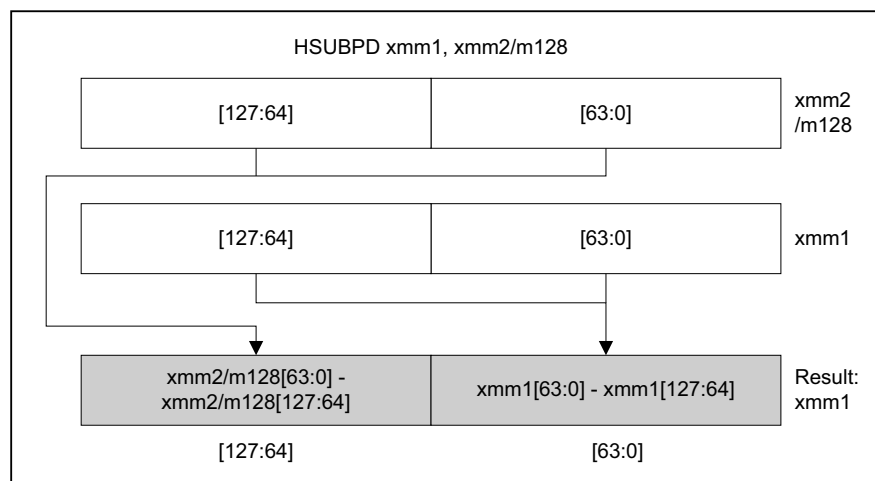
The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-19 for HSUBPD; see Figure 3-20 for VHSUBPD.



OM15995

Figure 3-19. HSUBPD—Packed Double-FP Horizontal Subtract

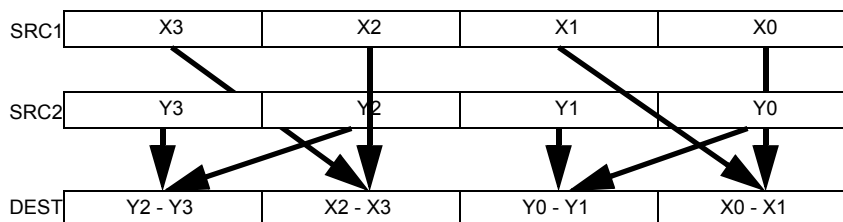


Figure 3-20. VHSUBPD operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HSUBPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VHSUBPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VHSUBPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC1[255:192]
DEST[255:192] ← SRC2[191:128] - SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPD: `__m128d _mm_hsub_pd(__m128d a, __m128d b)`

VHSUBPD: `__m256d _mm256_hsub_pd (__m256d a, __m256d b);`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

HSUBPS—Packed Single-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7D /r HSUBPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Horizontal subtract packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7D /r VHSUBPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7D /r VHSUBPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>r, w</i>)	ModRM:r/m (<i>r</i>)	NA	NA
RVM	ModRM:reg (<i>w</i>)	VEX.vvvv (<i>r</i>)	ModRM:r/m (<i>r</i>)	NA

Description

Subtracts the single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15). See Figure 3-21 for HSUBPS; see Figure 3-22 for VHSUBPS.

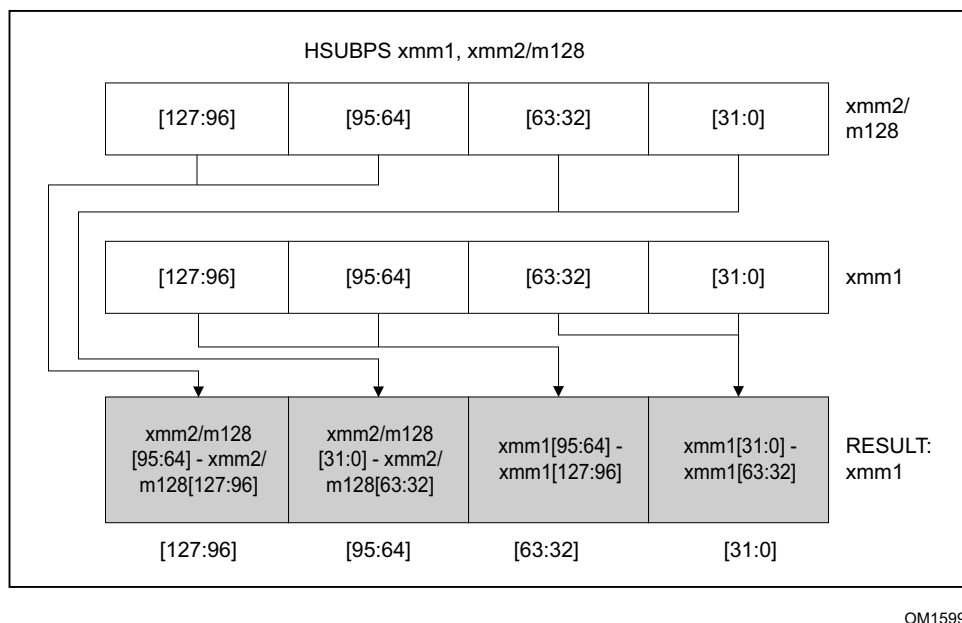


Figure 3-21. HSUBPS—Packed Single-FP Horizontal Subtract

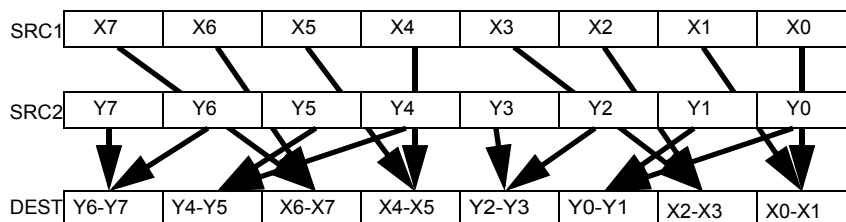


Figure 3-22. VHSUBPS operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HSUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128] \text{ (Unmodified)}$

VHSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VHSUBPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224]$
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[159:128] - \text{SRC2}[191:160]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[223:192] - \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS: `__m128 _mm_hsub_ps(__m128 a, __m128 b);`

VHSUBPS: `__m256 _mm256_hsub_ps (__m256 a, __m256 b);`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

IDIV—Signed Divide

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /7	IDIV <i>r/m8</i>	M	Valid	Valid	Signed divide AX by <i>r/m8</i> , with result stored in: AL ← Quotient, AH ← Remainder.
REX + F6 /7	IDIV <i>r/m8</i> *	M	Valid	N.E.	Signed divide AX by <i>r/m8</i> , with result stored in: AL ← Quotient, AH ← Remainder.
F7 /7	IDIV <i>r/m16</i>	M	Valid	Valid	Signed divide DX:AX by <i>r/m16</i> , with result stored in: AX ← Quotient, DX ← Remainder.
F7 /7	IDIV <i>r/m32</i>	M	Valid	Valid	Signed divide EDX:EAX by <i>r/m32</i> , with result stored in: EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /7	IDIV <i>r/m64</i>	M	Valid	N.E.	Signed divide RDX:RAX by <i>r/m64</i> , with result stored in: RAX ← Quotient, RDX ← Remainder.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (<i>r</i>)	NA	NA	NA

Description

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-59.

Table 3-59. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	<i>r/m8</i>	AL	AH	−128 to +127
Doubleword/word	DX:AX	<i>r/m16</i>	AX	DX	−32,768 to +32,767
Quadword/doubleword	EDX:EAX	<i>r/m32</i>	EAX	EDX	−2 ³¹ to 2 ³¹ − 1
Doublequadword/ quadword	RDX:RAX	<i>r/m64</i>	RAX	RDX	−2 ⁶³ to 2 ⁶³ − 1

Operation

```

IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp ← AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
            (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
                (* If a positive result is greater than 7FFFH
                or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX ← temp;
                    DX ← DX:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ← EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFH) or (temp < 80000000H)
                (* If a positive result is greater than 7FFFFFFFFFH
                or a negative result is less than 80000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    EAX ← temp;
                    EDX ← EDX:EAX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
                (* If a positive result is greater than 7FFFFFFFFFFFFFFFFFH
                or a negative result is less than 8000000000000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    RAX ← temp;
                    RDX ← RDX:RAX SignedModulus SRC;
            FI;
        FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#GP(0)	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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#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.
#UD	If the LOCK prefix is used.

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL <i>r/m8</i> *	M	Valid	Valid	AX ← AL * <i>r/m</i> byte.
F7 /5	IMUL <i>r/m16</i>	M	Valid	Valid	DX:AX ← AX * <i>r/m</i> word.
F7 /5	IMUL <i>r/m32</i>	M	Valid	Valid	EDX:EAX ← EAX * <i>r/m32</i> .
REX.W + F7 /5	IMUL <i>r/m64</i>	M	Valid	N.E.	RDX:RAX ← RAX * <i>r/m64</i> .
0F AF /r	IMUL <i>r16, r/m16</i>	RM	Valid	Valid	word register ← word register * <i>r/m16</i> .
0F AF /r	IMUL <i>r32, r/m32</i>	RM	Valid	Valid	doubleword register ← doubleword register * <i>r/m32</i> .
REX.W + 0F AF /r	IMUL <i>r64, r/m64</i>	RM	Valid	N.E.	Quadword register ← Quadword register * <i>r/m64</i> .
6B /r ib	IMUL <i>r16, r/m16, imm8</i>	RMI	Valid	Valid	word register ← <i>r/m16</i> * sign-extended immediate byte.
6B /r ib	IMUL <i>r32, r/m32, imm8</i>	RMI	Valid	Valid	doubleword register ← <i>r/m32</i> * sign-extended immediate byte.
REX.W + 6B /r ib	IMUL <i>r64, r/m64, imm8</i>	RMI	Valid	N.E.	Quadword register ← <i>r/m64</i> * sign-extended immediate byte.
69 /r iw	IMUL <i>r16, r/m16, imm16</i>	RMI	Valid	Valid	word register ← <i>r/m16</i> * immediate word.
69 /r id	IMUL <i>r32, r/m32, imm32</i>	RMI	Valid	Valid	doubleword register ← <i>r/m32</i> * immediate doubleword.
REX.W + 69 /r id	IMUL <i>r64, r/m64, imm32</i>	RMI	Valid	N.E.	Quadword register ← <i>r/m64</i> * immediate doubleword.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (<i>r, w</i>)	NA	NA	NA
RM	ModRM:reg (<i>r, w</i>)	ModRM:r/m (<i>r</i>)	NA	NA
RMI	ModRM:reg (<i>r, w</i>)	ModRM:r/m (<i>r</i>)	imm8/16/32	NA

Description

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

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).

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

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

Operation

```

IF (NumberOfOperands = 1)
  THEN IF (OperandSize = 8)
    THEN
      TMP_XP ← AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
      AX ← TMP_XP[15:0];
      SF ← TMP_XP[7];
      IF SignExtend(TMP_XP[7:0]) = TMP_XP
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; FI;
    ELSE IF OperandSize = 16
      THEN
        TMP_XP ← AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DX:AX ← TMP_XP[31:0];
        SF ← TMP_XP[15];
        IF SignExtend(TMP_XP[15:0]) = TMP_XP
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; FI;
    ELSE IF OperandSize = 32
      THEN
        TMP_XP ← EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        EDX:EAX ← TMP_XP[63:0];
        SF ← TMP_XP[32];
        IF SignExtend(TMP_XP[31:0]) = TMP_XP
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; FI;
    ELSE (* OperandSize = 64 *)
      TMP_XP ← RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
      EDX:EAX ← TMP_XP[127:0];
      SF ← TMP_XP[63];

```

```

        IF SignExtend(TMP_XP[63:0]) = TMP_XP
            THEN CF ← 0; OF ← 0;
            ELSE CF ← 1; OF ← 1; FI;
    FI;
ELSE IF (NumberOfOperands = 2)
    THEN
        TMP_XP ← DEST * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DEST ← TruncateToOperandSize(TMP_XP);
        SF ← MSB(DEST);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF ← 1; OF ← 1;
            ELSE CF ← 0; OF ← 0; FI;
    ELSE (* NumberOfOperands = 3 *)
        TMP_XP ← SRC1 * SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
        DEST ← TruncateToOperandSize(TMP_XP);
        SF ← MSB(DEST);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF ← 1; OF ← 1;
            ELSE CF ← 0; OF ← 0; FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

IN—Input from Port

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E4 <i>ib</i>	IN AL, <i>imm8</i>	I	Valid	Valid	Input byte from <i>imm8</i> I/O port address into AL.
E5 <i>ib</i>	IN AX, <i>imm8</i>	I	Valid	Valid	Input word from <i>imm8</i> I/O port address into AX.
E5 <i>ib</i>	IN EAX, <i>imm8</i>	I	Valid	Valid	Input dword from <i>imm8</i> I/O port address into EAX.
EC	IN AL,DX	NP	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	NP	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	NP	Valid	Valid	Input doubleword from I/O port in DX into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	<i>imm8</i>	NA	NA	NA
NP	NA	NA	NA	NA

Description

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

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 16, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Read from selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Read from selected I/O port *)
  FI;

```

Flags Affected

None.

Protected Mode Exceptions

- | | |
|--------|--|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. |
| #UD | If the LOCK prefix is used. |

Real-Address Mode Exceptions

- | | |
|-----|-----------------------------|
| #UD | If the LOCK prefix is used. |
|-----|-----------------------------|

Virtual-8086 Mode Exceptions

- | | |
|-----------------|--|
| #GP(0) | If any of the I/O permission bits in the TSS for the I/O port being accessed is 1. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- | | |
|--------|--|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. |
| #UD | If the LOCK prefix is used. |

INC—Increment by 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FE /0	INC <i>r/m8</i>	M	Valid	Valid	Increment <i>r/m</i> byte by 1.
REX + FE /0	INC <i>r/m8</i> *	M	Valid	N.E.	Increment <i>r/m</i> byte by 1.
FF /0	INC <i>r/m16</i>	M	Valid	Valid	Increment <i>r/m</i> word by 1.
FF /0	INC <i>r/m32</i>	M	Valid	Valid	Increment <i>r/m</i> doubleword by 1.
REX.W + FF /0	INC <i>r/m64</i>	M	Valid	N.E.	Increment <i>r/m</i> quadword by 1.
40+ <i>rw</i> **	INC <i>r16</i>	O	N.E.	Valid	Increment word register by 1.
40+ <i>rd</i>	INC <i>r32</i>	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	NA	NA	NA
O	opcode + <i>rd</i> (<i>r</i> , <i>w</i>)	NA	NA	NA

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.

In 64-bit mode, INC *r16* and INC *r32* are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

Operation

DEST ← DEST + 1;

AFlags 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 non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6C	INS <i>m8</i> , DX	NP	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.*
6D	INS <i>m16</i> , DX	NP	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INS <i>m32</i> , DX	NP	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6C	INSB	NP	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI. ¹
6D	INSW	NP	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INSD	NP	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹

NOTES:

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

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

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

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

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ /REPNE/REPZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 16, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Read from I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL IOPL *)
        DEST ← SRC; (* Read from I/O port *)
    FI;
```

Non-64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E)DI ← (E)DI + 1;
        ELSE (E)DI ← (E)DI - 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI - 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI - 4; FI;
        FI;
    FI;
```

FI64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E|R)DI ← (E|R)DI + 1;
        ELSE (E|R)DI ← (E|R)DI - 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI - 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E|R)DI ← (E|R)DI + 4;
                ELSE (E|R)DI ← (E|R)DI - 4; FI;
        FI;
```

FI;

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. If the destination is located in a non-writable segment. If an illegal memory operand effective address in the ES segments is given.
#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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

INSERTPS — Insert Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm2/m32</i> into <i>xmm1</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Insert a single precision floating point value selected by <i>imm8</i> from <i>xmm3/m32</i> and merge into <i>xmm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(register source form)

Select a single precision floating-point element from second source as indicated by Count_S bits of the immediate operand and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

Load a floating-point element from a 32-bit memory location and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version. The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

INSERTPS (128-bit Legacy SSE version)

IF (SRC = REG) THEN COUNT_S \leftarrow imm8[7:6]

ELSE COUNT_S \leftarrow 0

COUNT_D \leftarrow imm8[5:4]

ZMASK \leftarrow imm8[3:0]

CASE (COUNT_S) OF

0: TMP \leftarrow SRC[31:0]

1: TMP \leftarrow SRC[63:32]

2: TMP \leftarrow SRC[95:64]

3: TMP \leftarrow SRC[127:96]

ESAC;

CASE (COUNT_D) OF

0: TMP2[31:0] \leftarrow TMP

TMP2[127:32] \leftarrow DEST[127:32]

1: TMP2[63:32] \leftarrow TMP

TMP2[31:0] \leftarrow DEST[31:0]

TMP2[127:64] \leftarrow DEST[127:64]

2: TMP2[95:64] \leftarrow TMP

TMP2[63:0] \leftarrow DEST[63:0]

TMP2[127:96] \leftarrow DEST[127:96]

3: TMP2[127:96] \leftarrow TMP

TMP2[95:0] \leftarrow DEST[95:0]

ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] \leftarrow 00000000H

ELSE DEST[31:0] \leftarrow TMP2[31:0]

IF (ZMASK[1] = 1) THEN DEST[63:32] \leftarrow 00000000H

ELSE DEST[63:32] \leftarrow TMP2[63:32]

IF (ZMASK[2] = 1) THEN DEST[95:64] \leftarrow 00000000H

ELSE DEST[95:64] \leftarrow TMP2[95:64]

IF (ZMASK[3] = 1) THEN DEST[127:96] \leftarrow 00000000H

ELSE DEST[127:96] \leftarrow TMP2[127:96]

DEST[VLMAX-1:128] (Unmodified)

VINSERTPS (VEX.128 encoded version)

IF (SRC = REG) THEN COUNT_S \leftarrow imm8[7:6]

ELSE COUNT_S \leftarrow 0

COUNT_D \leftarrow imm8[5:4]

ZMASK \leftarrow imm8[3:0]

CASE (COUNT_S) OF

0: TMP \leftarrow SRC2[31:0]

1: TMP \leftarrow SRC2[63:32]

2: TMP \leftarrow SRC2[95:64]

3: TMP \leftarrow SRC2[127:96]

ESAC;

CASE (COUNT_D) OF

0: TMP2[31:0] \leftarrow TMP

TMP2[127:32] \leftarrow SRC1[127:32]

1: TMP2[63:32] \leftarrow TMP

TMP2[31:0] \leftarrow SRC1[31:0]

TMP2[127:64] \leftarrow SRC1[127:64]

```

2: TMP2[95:64] ← TMP
   TMP2[63:0] ← SRC1[63:0]
   TMP2[127:96] ← SRC1[127:96]
3: TMP2[127:96] ← TMP
   TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
ELSE DEST[127:96] ← TMP2[127:96]
DEST[VLMAX-1:128] ← 0

```

Intel C/C++ Compiler Intrinsic Equivalent

INSERTPS: `__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5.

INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CC	INT 3	NP	Valid	Valid	Interrupt 3—trap to debugger.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	NP	Invalid	Valid	Interrupt 4—if overflow flag is 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	imm8	NA	NA	NA

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

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

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

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

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

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

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the “Operation” section for this instruction (except #GP).

Table 3-60. Decision Table

PE	0	1	1	1	1	1	1	1
----	---	---	---	---	---	---	---	---

Table 3-60. Decision Table (Contd.)

VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	=3
DPL/CPL RELATIONSHIP	-	DPL < CPL	-	DPL > CPL	DPL = CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

- Don't Care.
- Y Yes, action taken.
- Blank Action not taken.

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

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

Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction `error_code(num,idt,ext)`, where `idt` and `ext` are bit values. The pseudofunction produces an error code as follows: (1) if `idt` is 0, the error code is $(\text{num} \& \text{FCH}) \mid \text{ext}$; (2) if `idt` is 1, the error code is $(\text{num} \ll 3) \mid 2 \mid \text{ext}$.

In many cases, the pseudofunction `error_code` is invoked with a pseudovisible EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt, EXT is 0; otherwise, EXT is 1.

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (VM = 1 and IOPL < 3 AND INT n)
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT n *)
      ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
        IF (IA32_EFER.LMA = 0)
          THEN (* Protected mode, or virtual-8086 mode interrupt *)
            GOTO PROTECTED-MODE;
          ELSE (* IA-32e mode interrupt *)
            GOTO IA-32e-MODE;
        FI;
      FI;
    FI;
  FI;
REAL-ADDRESS-MODE:
  IF ((vector_number << 2) + 3) is not within IDT limit
    THEN #GP; FI;
  IF stack not large enough for a 6-byte return information
    THEN #SS; FI;
  Push (EFLAGS[15:0]);
  IF ← 0; (* Clear interrupt flag *)
  TF ← 0; (* Clear trap flag *)
  AC ← 0; (* Clear AC flag *)
  Push(CS);
  Push(IP);
  (* No error codes are pushed in real-address mode*)
  CS ← IDT(Descriptor (vector_number << 2), selector));
  EIP ← IDT(Descriptor (vector_number << 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
  IF ((vector_number << 3) + 7) is not within IDT limits
  or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
    THEN #GP(error_code(vector_number,1,EXT)); FI;
    (* idt operand to error_code set because vector is used *)
  IF software interrupt (* Generated by INT n, INT3, or INTO *)
    THEN
      IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
        THEN #GP(error_code(vector_number,1,0)); FI;
        (* idt operand to error_code set because vector is used *)
        (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
      FI;
    IF gate not present
      THEN #NP(error_code(vector_number,1,EXT)); FI;
      (* idt operand to error_code set because vector is used *)
    IF task gate (* Specified in the selected interrupt table descriptor *)
      THEN GOTO TASK-GATE;
    ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
  FI;
END;
IA-32e-MODE:
  IF INTO and CS.L = 1 (64-bit mode)
    THEN #UD;

```

```

FI;
IF ((vector_number << 4) + 15) is not in IDT limits
or selected IDT descriptor is not an interrupt-, or trap-gate type
    THEN #GP(error_code(vector_number,1,EXT));
    (* idt operand to error_code set because vector is used *)
FI;
IF software interrupt (* Generated by INT n, INT 3, or INTO *)
    THEN
        IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
            THEN #GP(error_code(vector_number,1,0));
            (* idt operand to error_code set because vector is used *)
            (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
        FI;
    FI;
IF gate not present
    THEN #NP(error_code(vector_number,1,EXT));
    (* idt operand to error_code set because vector is used *)
FI;
GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
    IF local/global bit is set to local or index not within GDT limits
        THEN #GP(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(EXT); FI;
            Push(error code);
        FI;
    IF EIP not within code segment limit
        THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,

```

```

    THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new code segment is non-conforming with DPL < CPL
    THEN
        IF VM = 0
            THEN
                GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                DPL < CPL *)
            ELSE (* VM = 1 *)
                IF new code-segment DPL ≠ 0
                    THEN #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
                FI;
            ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
                IF VM = 1
                    THEN #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
                IF new code segment is conforming or new code-segment DPL = CPL
                    THEN
                        GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                    ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                        #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    FI;
                FI;
            END;
        INTER-PRIVILEGE-LEVEL-INTERRUPT:
        (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
        IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
            THEN
                (* Identify stack-segment selector for new privilege level in current TSS *)
                IF current TSS is 32-bit
                    THEN
                        TSSstackAddress ← (new code-segment DPL << 3) + 4;
                        IF (TSSstackAddress + 5) > current TSS limit
                            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                            (* idt operand to error_code is 0 because selector is used *)
                            NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                            NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
                        ELSE (* current TSS is 16-bit *)
                            TSSstackAddress ← (new code-segment DPL << 2) + 2
                            IF (TSSstackAddress + 3) > current TSS limit
                                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                                (* idt operand to error_code is 0 because selector is used *)
                                NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                                NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
                            FI;
                        IF NewSS is NULL
                            THEN #TS(EXT); FI;
                        IF NewSS index is not within its descriptor-table limits
                            or NewSS RPL ≠ new code-segment DPL

```



```

        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read new stack-segment descriptor for NewSS in GDT or LDT;
    IF new stack-segment DPL ≠ new code-segment DPL
    or new stack-segment Type does not indicate writable data segment
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF NewSS is not present
        THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IA-32e mode *)
        IF IDT-gate IST = 0
            THEN TSSstackAddress ← (new code-segment DPL << 3) + 4;
            ELSE TSSstackAddress ← (IDT gate IST << 3) + 28;
        FI;
        IF (TSSstackAddress + 7) > current TSS limit
            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
    FI;
    IF IDT gate is 32-bit
        THEN
            IF new stack does not have room for 24 bytes (error code pushed)
            or 20 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            FI
        ELSE
            IF IDT gate is 16-bit
                THEN
                    IF new stack does not have room for 12 bytes (error code pushed)
                    or 10 bytes (no error code pushed);
                        THEN #SS(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    ELSE (* 64-bit IDT gate*)
                        IF StackAddress is non-canonical
                            THEN #SS(EXT); FI; (* Error code contains NULL selector *)
                        FI;
                    FI;
                IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
                    THEN
                        IF instruction pointer from IDT gate is not within new code-segment limits
                            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                        ESP ← NewESP;
                        SS ← NewSS; (* Segment descriptor information also loaded *)
                    ELSE (* IA-32e mode *)
                        IF instruction pointer from IDT gate contains a non-canonical address
                            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                        RSP ← NewRSP & FFFFFFFF0H;
                        SS ← NewSS;
                    FI;
                IF IDT gate is 32-bit
                    THEN

```

```

    CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
ELSE
    IF IDT gate 16-bit
    THEN
        CS:IP ← Gate(CS:IP);
        (* Segment descriptor information also loaded *)
    ELSE (* 64-bit IDT gate *)
        CS:RIP ← Gate(CS:RIP);
        (* Segment descriptor information also loaded *)
    FI;
FI;
IF IDT gate is 32-bit
THEN
    Push(far pointer to old stack);
    (* Old SS and ESP, 3 words padded to 4 *)
    Push(EFLAGS);
    Push(far pointer to return instruction);
    (* Old CS and EIP, 3 words padded to 4 *)
    Push(ErrorCode); (* If needed, 4 bytes *)
ELSE
    IF IDT gate 16-bit
    THEN
        Push(far pointer to old stack);
        (* Old SS and SP, 2 words *)
        Push(EFLAGS(15-0));
        Push(far pointer to return instruction);
        (* Old CS and IP, 2 words *)
        Push(ErrorCode); (* If needed, 2 bytes *)
    ELSE (* 64-bit IDT gate *)
        Push(far pointer to old stack);
        (* Old SS and SP, each an 8-byte push *)
        Push(RFLAGS); (* 8-byte push *)
        Push(far pointer to return instruction);
        (* Old CS and RIP, each an 8-byte push *)
        Push(ErrorCode); (* If needed, 8-bytes *)
    FI;
FI;
CPL ← new code-segment DPL;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
TF ← 0;
VM ← 0;
RF ← 0;
NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
(* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
THEN
    IF TSS limit < 9
    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
    NewSS ← 2 bytes loaded from (current TSS base + 8);

```

```

    NewESP ← 4 bytes loaded from (current TSS base + 4);
ELSE (* current TSS is 16-bit *)
    IF TSS limit < 5
        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    NewSS ← 2 bytes loaded from (current TSS base + 4);
    NewESP ← 2 bytes loaded from (current TSS base + 2);
FI;
IF NewSS is NULL
    THEN #TS(EXT); FI; (* Error code contains NULL selector *)
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
    THEN #SS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 40 bytes (error code pushed)
        or 36 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        ELSE (* IDT gate is 16-bit)
            IF new stack does not have room for 20 bytes (error code pushed)
            or 18 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
        FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
    THEN IF = 0; FI;
TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);

```

```

Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS:IP ← Gate(CS); (* Segment descriptor information also loaded *)
IF OperandSize = 32
    THEN
        EIP ← Gate(instruction pointer);
    ELSE (* OperandSize is 16 *)
        EIP ← Gate(instruction pointer) AND 0000FFFFH;
FI;
(* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
    IF IDT-descriptor IST ≠ 0
        THEN
            TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
    IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
                or 12 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
            ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
                IF current stack does not have room for 8 bytes (error code pushed)
                    or 6 bytes (no error code pushed)
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
                ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                    IF NewRSP contains a non-canonical address
                        THEN #SS(EXT); (* Error code contains NULL selector *)
                FI;
            FI;
        IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
            THEN
                IF instruction pointer from IDT gate is not within new code-segment limit
                    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            ELSE
                IF instruction pointer from IDT gate contains a non-canonical address
                    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                RSP ← NewRSP & FFFFFFFF0H;
            FI;
        IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
            THEN
                Push (EFLAGS);
                Push (far pointer to return instruction); (* 3 words padded to 4 *)

```

```

    CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
    Push (ErrorCode); (* If any *)
ELSE
    IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
    THEN
        Push (FLAGS);
        Push (far pointer to return location); (* 2 words *)
        CS:IP ← Gate(CS:IP);
        (* Segment descriptor information also loaded *)
        Push (ErrorCode); (* If any *)
    ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
        Push(far pointer to old stack);
        (* Old SS and SP, each an 8-byte push *)
        Push(RFLAGS); (* 8-byte push *)
        Push(far pointer to return instruction);
        (* Old CS and RIP, each an 8-byte push *)
        Push(ErrorCode); (* If needed, 8 bytes *)
        CS:RIP ← GATE(CS:RIP);
        (* Segment descriptor information also loaded *)
    FI;
FI;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
    THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
TF ← 0;
NT ← 0;
VM ← 0;
RF ← 0;
END;

```

Flags Affected

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

Protected Mode Exceptions

- | | |
|-----------------|--|
| #GP(error_code) | <p>If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt-, trap-, or task gate is NULL.</p> <p>If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.</p> <p>If an interrupt is generated by the INT <i>n</i>, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.</p> <p>If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p> |
| #SS(error_code) | <p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p> <p>If the SS register is being loaded and the segment pointed to is marked not present.</p> |

	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. If the stack segment selector in the TSS is NULL. If the stack segment for the TSS is not a writable data segment. If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push. If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT <i>n</i> , INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3. If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits. If the segment selector in the interrupt-, trap-, or task gate is NULL. If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits. If the vector selects a descriptor outside the IDT limits. If an IDT descriptor is not an interrupt-, trap-, or task-descriptor. If an interrupt is generated by the INT <i>n</i> instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL. If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment. If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present. If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. If the stack segment selector in the TSS is NULL. If the stack segment for the TSS is not a writable data segment. If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.

#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code)	<p>If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.</p> <p>If the segment selector in the 64-bit interrupt or trap gate is NULL.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If the vector points to a gate which is in non-canonical space.</p> <p>If the vector points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.</p> <p>If the descriptor pointed to by the gate selector is outside the descriptor table limit.</p> <p>If the descriptor pointed to by the gate selector is in non-canonical space.</p> <p>If the descriptor pointed to by the gate selector is not a code segment.</p> <p>If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.</p> <p>If the descriptor pointed to by the gate selector has DPL > CPL.</p>
#SS(error_code)	<p>If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.</p> <p>If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).</p>
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	<p>If an attempt to load RSP from the TSS causes an access to non-canonical space.</p> <p>If the RSP from the TSS is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

INVD—Invalidate Internal Caches

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 08	INVD	NP	Valid	Valid	Flush internal caches; initiate flushing of external caches.

NOTES:

* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

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

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The INVD instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

INVLPG—Invalidate TLB Entries

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01/7	INVLPG <i>m</i>	M	Valid	Valid	Invalidate TLB entries for page containing <i>m</i> .

NOTES:

* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.¹

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

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidate only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see “MOV—Move to/from Control Registers” and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

This instruction’s operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

Invalidate(RelevantTLBEntries);
Continue; (* Continue execution *)

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD Operand is a register.
If the LOCK prefix is used.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), the instruction invalidates all of them.

Real-Address Mode Exceptions

#UD Operand is a register.
 If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The INVLPG instruction cannot be executed at the virtual-8086 mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD Operand is a register.
 If the LOCK prefix is used.

INVPCID—Invalidate Process-Context Identifier

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 82 /r INVPCID r32, m128	RM	NE/V	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 0F 38 82 /r INVPCID r64, m128	RM	V/NE	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (R)	ModRM:r/m (R)	NA	NA

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, “Caching Translation Information,” in *Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A*.) Invalidation is based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits.

There are four INVPCID types currently defined:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor.¹ In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- All-context invalidation, including global translations: If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- All-context invalidation: If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-23. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), the instruction invalidates all of them.

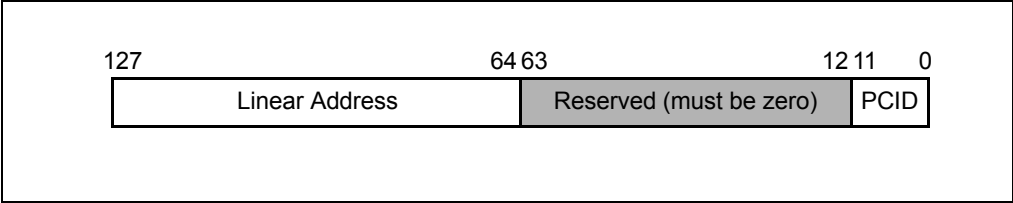


Figure 3-23. INVPCID Descriptor

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H; executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside 64-bit mode (see Chapter 4.10.1, “Process-Context Identifiers (PCIDs),” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Operation

```
INVPCID_TYPE ← value of register operand;      // must be in the range of 0-3
INVPCID_DESC ← value of memory operand;
CASE INVPCID_TYPE OF
  0:      // individual-address invalidation
    PCID ← INVPCID_DESC[11:0];
    L_ADDR ← INVPCID_DESC[127:64];
    Invalidate mappings for L_ADDR associated with PCID except global translations;
    BREAK;
  1:      // single PCID invalidation
    PCID ← INVPCID_DESC[11:0];
    Invalidate all mappings associated with PCID except global translations;
    BREAK;
  2:      // all PCID invalidation including global translations
    Invalidate all mappings for all PCIDs, including global translations;
    BREAK;
  3:      // all PCID invalidation retaining global translations
    Invalidate all mappings for all PCIDs except global translations;
    BREAK;
ESAC;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
INVPCID: void _invpcid(unsigned __int32 type, void * descriptor);
```

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains an unusable segment.
- If the source operand is located in an execute-only code segment.
- If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
- If bits 63:12 of INVPCID_DESC are not all zero.

	If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#UD	If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The INVPCID instruction is not recognized in virtual-8086 mode.
--------	---

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.

IRET/IRETD—Interrupt Return

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CF	IRET	NP	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	NP	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	NP	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

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

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

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

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

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

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

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, “Handling Multiple NMIs” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), execution of the IRET instruction unblocks NMIs.

This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```

IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE:
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
            EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop();
        FI;
    END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS ← Pop();
                (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            FI;
        ELSE

```



```

        #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
        FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS ← tempEFLAGS;
    ESP ← Pop();
    SS ← Pop(); (* Pop 2 words; throw away high-order word *)
    ES ← Pop(); (* Pop 2 words; throw away high-order word *)
    DS ← Pop(); (* Pop 2 words; throw away high-order word *)
    FS ← Pop(); (* Pop 2 words; throw away high-order word *)
    GS ← Pop(); (* Pop 2 words; throw away high-order word *)
    CPL ← 3;
    (* Resume execution in Virtual-8086 mode *)
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF new mode ≠ 64-Bit Mode
        THEN

```

```

        IF EIP is not within CS limit
            THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                THEN #GP(0); FI;
FI;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize = 32
    THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
IF CPL ≤ IOPL
    THEN EFLAGS(IF) ← tempEFLAGS; FI;
IF CPL = 0
    THEN
        EFLAGS(IOPL) ← tempEFLAGS;
        IF OperandSize = 32
            THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; FI;
        IF OperandSize = 64
            THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
FI;
CPL ← CS(RPL);
FOR each SegReg in (ES, FS, GS, and DS)
    DO
        tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
        IF tempDesc(DPL) < CPL AND tempDesc(Type) is data or non-conforming code
            THEN (* Segment register invalid *)
                SegReg ← NULL;
FI;
    OD;
END;

```

```

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
    IF new mode ≠ 64-Bit Mode
        THEN
            IF EIP is not within CS limit
                THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
                IF RIP is non-canonical
                    THEN #GP(0); FI;
FI;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize = 32 or OperandSize = 64
    THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
IF CPL ≤ IOPL
    THEN EFLAGS(IF) ← tempEFLAGS; FI;
IF CPL = 0
    THEN (* VM = 0 in flags image *)
        EFLAGS(IOPL) ← tempEFLAGS;
        IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
FI;
END;

```

```

IA-32e-MODE:
    IF NT = 1

```

```

    THEN #GP(0);
ELSE IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop();
        tempEFLAGS ← Pop();
    ELSE IF OperandSize = 16
        THEN
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
        FI;
    ELSE (* OperandSize = 64 *)
        THEN
            RIP ← Pop();
            CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
            tempRFLAGS ← Pop();
        FI;
    IF tempCS.RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
    ELSE
        IF instruction began in 64-Bit Mode
            THEN
                IF OperandSize = 32
                    THEN
                        ESP ← Pop();
                        SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                    ELSE IF OperandSize = 16
                        THEN
                            ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                            SS ← Pop(); (* 16-bit pop *)
                        ELSE (* OperandSize = 64 *)
                            RSP ← Pop();
                            SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                        FI;
                    FI;
                GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
    END;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	<p>If the return code or stack segment selector is NULL.</p> <p>If the return instruction pointer is not within the return code segment limit.</p>
#GP(selector)	<p>If a segment selector index is outside its descriptor table limits.</p> <p>If the return code segment selector RPL is less than the CPL.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p>

If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.

If the stack segment is not a writable data segment.

If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

If the segment descriptor for a code segment does not indicate it is a code segment.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is not busy.

If a TSS segment descriptor specifies that the TSS is not available.

#SS(0) If the top bytes of stack are not within stack limits.

#NP(selector) If the return code or stack segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the return instruction pointer is not within the return code segment limit.

#SS If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0) If the return instruction pointer is not within the return code segment limit.

IF IOPL not equal to 3.

#PF(fault-code) If a page fault occurs.

#SS(0) If the top bytes of stack are not within stack limits.

#AC(0) If an unaligned memory reference occurs and alignment checking is enabled.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.

If the return code segment selector is NULL.

If the stack segment selector is NULL going back to compatibility mode.

If the stack segment selector is NULL going back to CPL3 64-bit mode.

If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.

If the return instruction pointer is not within the return code segment limit.

If the return instruction pointer is non-canonical.

#GP(Selector) If a segment selector index is outside its descriptor table limits.

If a segment descriptor memory address is non-canonical.

If the segment descriptor for a code segment does not indicate it is a code segment.

If the proposed new code segment descriptor has both the D-bit and L-bit set.

If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.

If CPL is greater than the RPL of the code segment selector.

If the DPL of a conforming-code segment is greater than the return code segment selector RPL.

	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 <i>cb</i>	<i>JA rel8</i>	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 <i>cb</i>	<i>JAE rel8</i>	D	Valid	Valid	Jump short if above or equal (CF=0).
72 <i>cb</i>	<i>JB rel8</i>	D	Valid	Valid	Jump short if below (CF=1).
76 <i>cb</i>	<i>JBE rel8</i>	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 <i>cb</i>	<i>JC rel8</i>	D	Valid	Valid	Jump short if carry (CF=1).
E3 <i>cb</i>	<i>JCXZ rel8</i>	D	N.E.	Valid	Jump short if CX register is 0.
E3 <i>cb</i>	<i>JECXZ rel8</i>	D	Valid	Valid	Jump short if ECX register is 0.
E3 <i>cb</i>	<i>JRCXZ rel8</i>	D	Valid	N.E.	Jump short if RCX register is 0.
74 <i>cb</i>	<i>JE rel8</i>	D	Valid	Valid	Jump short if equal (ZF=1).
7F <i>cb</i>	<i>JG rel8</i>	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D <i>cb</i>	<i>JGE rel8</i>	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C <i>cb</i>	<i>JL rel8</i>	D	Valid	Valid	Jump short if less (SF≠ OF).
7E <i>cb</i>	<i>JLE rel8</i>	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 <i>cb</i>	<i>JNA rel8</i>	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 <i>cb</i>	<i>JNAE rel8</i>	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 <i>cb</i>	<i>JNB rel8</i>	D	Valid	Valid	Jump short if not below (CF=0).
77 <i>cb</i>	<i>JNBE rel8</i>	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 <i>cb</i>	<i>JNC rel8</i>	D	Valid	Valid	Jump short if not carry (CF=0).
75 <i>cb</i>	<i>JNE rel8</i>	D	Valid	Valid	Jump short if not equal (ZF=0).
7E <i>cb</i>	<i>JNG rel8</i>	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C <i>cb</i>	<i>JNGE rel8</i>	D	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D <i>cb</i>	<i>JNL rel8</i>	D	Valid	Valid	Jump short if not less (SF=OF).
7F <i>cb</i>	<i>JNLE rel8</i>	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 <i>cb</i>	<i>JNO rel8</i>	D	Valid	Valid	Jump short if not overflow (OF=0).
7B <i>cb</i>	<i>JNP rel8</i>	D	Valid	Valid	Jump short if not parity (PF=0).
79 <i>cb</i>	<i>JNS rel8</i>	D	Valid	Valid	Jump short if not sign (SF=0).
75 <i>cb</i>	<i>JNZ rel8</i>	D	Valid	Valid	Jump short if not zero (ZF=0).
70 <i>cb</i>	<i>JO rel8</i>	D	Valid	Valid	Jump short if overflow (OF=1).
7A <i>cb</i>	<i>JP rel8</i>	D	Valid	Valid	Jump short if parity (PF=1).
7A <i>cb</i>	<i>JPE rel8</i>	D	Valid	Valid	Jump short if parity even (PF=1).
7B <i>cb</i>	<i>JPO rel8</i>	D	Valid	Valid	Jump short if parity odd (PF=0).
78 <i>cb</i>	<i>JS rel8</i>	D	Valid	Valid	Jump short if sign (SF=1).
74 <i>cb</i>	<i>JZ rel8</i>	D	Valid	Valid	Jump short if zero (ZF = 1).
0F 87 <i>cw</i>	<i>JA rel16</i>	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 <i>cd</i>	<i>JA rel32</i>	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 <i>cw</i>	<i>JAE rel16</i>	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 83 <i>cd</i>	JAE <i>rel32</i>	D	Valid	Valid	Jump near if above or equal (CF=0).
0F 82 <i>cw</i>	JB <i>rel16</i>	D	N.S.	Valid	Jump near if below (CF=1). Not supported in 64-bit mode.
0F 82 <i>cd</i>	JB <i>rel32</i>	D	Valid	Valid	Jump near if below (CF=1).
0F 86 <i>cw</i>	JBE <i>rel16</i>	D	N.S.	Valid	Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 <i>cd</i>	JBE <i>rel32</i>	D	Valid	Valid	Jump near if below or equal (CF=1 or ZF=1).
0F 82 <i>cw</i>	JC <i>rel16</i>	D	N.S.	Valid	Jump near if carry (CF=1). Not supported in 64-bit mode.
0F 82 <i>cd</i>	JC <i>rel32</i>	D	Valid	Valid	Jump near if carry (CF=1).
0F 84 <i>cw</i>	JE <i>rel16</i>	D	N.S.	Valid	Jump near if equal (ZF=1). Not supported in 64-bit mode.
0F 84 <i>cd</i>	JE <i>rel32</i>	D	Valid	Valid	Jump near if equal (ZF=1).
0F 84 <i>cw</i>	JZ <i>rel16</i>	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 <i>cd</i>	JZ <i>rel32</i>	D	Valid	Valid	Jump near if 0 (ZF=1).
0F 8F <i>cw</i>	JG <i>rel16</i>	D	N.S.	Valid	Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F <i>cd</i>	JG <i>rel32</i>	D	Valid	Valid	Jump near if greater (ZF=0 and SF=OF).
0F 8D <i>cw</i>	JGE <i>rel16</i>	D	N.S.	Valid	Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.
0F 8D <i>cd</i>	JGE <i>rel32</i>	D	Valid	Valid	Jump near if greater or equal (SF=OF).
0F 8C <i>cw</i>	JL <i>rel16</i>	D	N.S.	Valid	Jump near if less (SF≠OF). Not supported in 64-bit mode.
0F 8C <i>cd</i>	JL <i>rel32</i>	D	Valid	Valid	Jump near if less (SF≠OF).
0F 8E <i>cw</i>	JLE <i>rel16</i>	D	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠OF). Not supported in 64-bit mode.
0F 8E <i>cd</i>	JLE <i>rel32</i>	D	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠OF).
0F 86 <i>cw</i>	JNA <i>rel16</i>	D	N.S.	Valid	Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 <i>cd</i>	JNA <i>rel32</i>	D	Valid	Valid	Jump near if not above (CF=1 or ZF=1).
0F 82 <i>cw</i>	JNAE <i>rel16</i>	D	N.S.	Valid	Jump near if not above or equal (CF=1). Not supported in 64-bit mode.
0F 82 <i>cd</i>	JNAE <i>rel32</i>	D	Valid	Valid	Jump near if not above or equal (CF=1).
0F 83 <i>cw</i>	JNB <i>rel16</i>	D	N.S.	Valid	Jump near if not below (CF=0). Not supported in 64-bit mode.
0F 83 <i>cd</i>	JNB <i>rel32</i>	D	Valid	Valid	Jump near if not below (CF=0).
0F 87 <i>cw</i>	JNBE <i>rel16</i>	D	N.S.	Valid	Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 <i>cd</i>	JNBE <i>rel32</i>	D	Valid	Valid	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 <i>cw</i>	JNC <i>rel16</i>	D	N.S.	Valid	Jump near if not carry (CF=0). Not supported in 64-bit mode.
0F 83 <i>cd</i>	JNC <i>rel32</i>	D	Valid	Valid	Jump near if not carry (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 85 <i>cw</i>	JNE <i>rel16</i>	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
0F 85 <i>cd</i>	JNE <i>rel32</i>	D	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E <i>cw</i>	JNG <i>rel16</i>	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF≠OF). Not supported in 64-bit mode.
0F 8E <i>cd</i>	JNG <i>rel32</i>	D	Valid	Valid	Jump near if not greater (ZF=1 or SF≠OF).
0F 8C <i>cw</i>	JNGE <i>rel16</i>	D	N.S.	Valid	Jump near if not greater or equal (SF≠OF). Not supported in 64-bit mode.
0F 8C <i>cd</i>	JNGE <i>rel32</i>	D	Valid	Valid	Jump near if not greater or equal (SF≠OF).
0F 8D <i>cw</i>	JNL <i>rel16</i>	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
0F 8D <i>cd</i>	JNL <i>rel32</i>	D	Valid	Valid	Jump near if not less (SF=OF).
0F 8F <i>cw</i>	JNLE <i>rel16</i>	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F <i>cd</i>	JNLE <i>rel32</i>	D	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 <i>cw</i>	JNO <i>rel16</i>	D	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
0F 81 <i>cd</i>	JNO <i>rel32</i>	D	Valid	Valid	Jump near if not overflow (OF=0).
0F 8B <i>cw</i>	JNP <i>rel16</i>	D	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
0F 8B <i>cd</i>	JNP <i>rel32</i>	D	Valid	Valid	Jump near if not parity (PF=0).
0F 89 <i>cw</i>	JNS <i>rel16</i>	D	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
0F 89 <i>cd</i>	JNS <i>rel32</i>	D	Valid	Valid	Jump near if not sign (SF=0).
0F 85 <i>cw</i>	JNZ <i>rel16</i>	D	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
0F 85 <i>cd</i>	JNZ <i>rel32</i>	D	Valid	Valid	Jump near if not zero (ZF=0).
0F 80 <i>cw</i>	JO <i>rel16</i>	D	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
0F 80 <i>cd</i>	JO <i>rel32</i>	D	Valid	Valid	Jump near if overflow (OF=1).
0F 8A <i>cw</i>	JP <i>rel16</i>	D	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
0F 8A <i>cd</i>	JP <i>rel32</i>	D	Valid	Valid	Jump near if parity (PF=1).
0F 8A <i>cw</i>	JPE <i>rel16</i>	D	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
0F 8A <i>cd</i>	JPE <i>rel32</i>	D	Valid	Valid	Jump near if parity even (PF=1).
0F 8B <i>cw</i>	JPO <i>rel16</i>	D	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.
0F 8B <i>cd</i>	JPO <i>rel32</i>	D	Valid	Valid	Jump near if parity odd (PF=0).
0F 88 <i>cw</i>	JS <i>rel16</i>	D	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 88 <i>cd</i>	<i>J</i> S <i>rel32</i>	D	Valid	Valid	Jump near if sign (SF=1).
0F 84 <i>cw</i>	<i>J</i> Z <i>rel16</i>	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 <i>cd</i>	<i>J</i> Z <i>rel32</i>	D	Valid	Valid	Jump near if 0 (ZF=1).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

Description

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

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

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

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

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

```
JZ FARLABEL;
```

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

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

The *JRCXZ*, *JECXZ* and *JCXZ* instructions differ from other *Jcc* instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as *LOOPNE*). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute 2^{64} , 2^{32} or 64K times (not zero times).

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

In 64-bit mode, operand size is fixed at 64 bits. *JMP* Short is $RIP = RIP + 8\text{-bit offset sign extended to 64 bits}$. *JMP* Near is $RIP = RIP + 32\text{-bit offset sign extended to 64-bits}$.

Operation

```

IF condition
  THEN
    tempEIP ← EIP + SignExtend(DEST);
    IF OperandSize = 16
      THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
  IF tempEIP is not within code segment limit
    THEN #GP(0);
    ELSE EIP ← tempEIP
  FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

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

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.

JMP—Jump

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
EB <i>cb</i>	JMP <i>rel8</i>	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 <i>cw</i>	JMP <i>rel16</i>	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 <i>cd</i>	JMP <i>rel32</i>	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP <i>r/m16</i>	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended <i>r/m16</i> . Not supported in 64-bit mode.
FF /4	JMP <i>r/m32</i>	M	N.S.	Valid	Jump near, absolute indirect, address given in <i>r/m32</i> . Not supported in 64-bit mode.
FF /4	JMP <i>r/m64</i>	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA <i>cd</i>	JMP <i>ptr16:16</i>	D	Inv.	Valid	Jump far, absolute, address given in operand
EA <i>cp</i>	JMP <i>ptr16:32</i>	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP <i>m16:16</i>	D	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:16</i>
FF /5	JMP <i>m16:32</i>	D	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:32</i> .
REX.W + FF /5	JMP <i>m16:64</i>	D	Valid	N.E.	Jump far, absolute indirect, address given in <i>m16:64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

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

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

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

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current

value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

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

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-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

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

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

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

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

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

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

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

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

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

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

In 64-Bit Mode — The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF near jump
  IF 64-bit Mode
    THEN
      IF near relative jump
        THEN
          tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempRIP ← DEST;
        FI;
      ELSE
        IF near relative jump
          THEN
            tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
          ELSE (* Near absolute jump *)
            tempEIP ← DEST;
          FI;
        FI;
      IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
        and tempEIP outside code segment limit
          THEN #GP(0); FI
      IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
      FI;
      IF OperandSize = 32
        THEN
          EIP ← tempEIP;
        ELSE
          IF OperandSize = 16
            THEN (* OperandSize = 16 *)
              EIP ← tempEIP AND 0000FFFFH;
            ELSE (* OperandSize = 64 *)
              RIP ← tempRIP;
            FI;
          FI;
        FI;
      IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
        THEN
          tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
          IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
          CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
          IF OperandSize = 32

```

```

        THEN
            EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
        ELSE (* OperandSize = 16 *)
            EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
    FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
            FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    ELSE
        #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
    tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;

```

NONCONFORMING-CODE-SEGMENT:

```

IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF (RPL > CPL) OR (DPL ≠ CPL)
    THEN #GP(code segment selector); FI;
IF segment not present
    THEN #NP(segment selector); FI;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
and tempEIP outside code segment limit
    THEN #GP(0); FI;
IF tempEIP is non-canonical THEN #GP(0); FI;
CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
END;
```

CALL-GATE:

```

IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); FI;
IF call gate not present
    THEN #NP(call gate selector); FI;
IF call gate code-segment selector is NULL
    THEN #GP(0); FI;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); FI;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
or code-segment segment descriptor is conforming and DPL > CPL
or code-segment segment descriptor is non-conforming and DPL ≠ CPL
    THEN #GP(code segment selector); FI;
IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
or code-segment segment descriptor has both L-Bit and D-bit set)
    THEN #GP(code segment selector); FI;
IF code segment is not present
    THEN #NP(code-segment selector); FI;
IF instruction pointer is not within code-segment limit
    THEN #GP(0); FI;
tempEIP ← DEST(Offset);
IF GateSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
outside code segment limit
    THEN #GP(0); FI;
CS ← DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
END;
```

TASK-GATE:

```

IF task gate DPL < CPL
or task gate DPL < task gate segment-selector RPL
```

```

    THEN #GP(task gate selector); FI;
IF task gate not present
    THEN #NP(gate selector); FI;
Read the TSS segment selector in the task-gate descriptor;
IF TSS segment selector local/global bit is set to local
or index not within GDT limits
or TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); FI;
IF TSS not present
    THEN #NP(TSS selector); FI;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL < CPL
    or TSS DPL < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	<p>If offset in target operand, call gate, or TSS is beyond the code segment limits.</p> <p>If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If the segment selector index is outside descriptor table limits.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL</p> <p>(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.</p> <p>If the segment descriptor for selector in a call gate does not indicate it is a code segment.</p> <p>If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p>

#NP (selector)	If the code segment being accessed is not present. If call gate, task gate, or TSS not present.
#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. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If the target operand is beyond the code segment limits. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical. If target offset in destination operand is non-canonical. If target offset in destination operand is beyond the new code segment limit. If the segment selector in the destination operand is NULL. If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If the code segment or 64-bit call gate is outside descriptor table limits. If the code segment or 64-bit call gate overlaps non-canonical space. If the segment descriptor from a 64-bit call gate is in non-canonical space. If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate. If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set. If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL. If the DPL for a conforming-code segment is greater than the CPL. If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate. If the upper type field of a 64-bit call gate is not 0x0. If the segment selector from a 64-bit call gate is beyond the descriptor table limits. If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear. If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.

	If the code segment is non-confirming and $CPL \neq DPL$.
	If the code segment is confirming and $CPL < DPL$.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#UD	(64-bit mode only) If a far jump is direct to an absolute address in memory.
	If the LOCK prefix is used.
#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.

LAHF—Load Status Flags into AH Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9F	LAHF	NP	Invalid*	Valid	Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

NOTES:
*Valid in specific steppings. See Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation

IF 64-Bit Mode
THEN
 IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
 THEN AH ← RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
 ELSE #UD;
 FI;
ELSE
 AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
FI;

Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.
 If the LOCK prefix is used.

LAR—Load Access Rights Byte

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 02 /r	LAR <i>r16</i> , <i>r16/m16</i>	RM	Valid	Valid	<i>r16</i> ← access rights referenced by <i>r16/m16</i>
OF 02 /r	LAR <i>reg</i> , <i>r32/m16</i> ¹	RM	Valid	Valid	<i>reg</i> ← access rights referenced by <i>r32/m16</i>

NOTES:

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

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

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
 - Bits 19:16 are undefined.
 - Bit 20 returns the software-available bit in the descriptor.
 - Bit 21 returns the L flag.
 - Bit 22 returns the D/B flag.
 - Bit 23 returns the G flag.
 - Bits 31:24 are returned as 0.

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

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-61.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

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

The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-61. Segment and Gate Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	No
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

IF Offset(SRC) > descriptor table limit

THEN

ZF ← 0;

ELSE

SegmentDescriptor ← descriptor referenced by SRC;

IF SegmentDescriptor(Type) ≠ conforming code segment

and (CPL > DPL) or (RPL > DPL)

or SegmentDescriptor(Type) is not valid for instruction

THEN

ZF ← 0;

ELSE

DEST ← access rights from SegmentDescriptor as given in Description section;

ZF ← 1;

FI;

FI;

Flags Affected

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

Protected Mode Exceptions

#GP(0) 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 the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LAR instruction is not recognized in real-address mode.
-----	---

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LDDQU—Load Unaligned Integer 128 Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F F0 /r LDDQU <i>xmm1</i> , <i>mem</i>	RM	V/V	SSE3	Load unaligned data from <i>mem</i> and return double quadword in <i>xmm1</i> .
VEX.128.F2.0F.WIG F0 /r VLDDQU <i>xmm1</i> , <i>m128</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>xmm1</i> .
VEX.256.F2.0F.WIG F0 /r VLDDQU <i>ymm1</i> , <i>m256</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The instruction is *functionally similar* to (V)MOVDQU *ymm/xmm*, *m256/m128* for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

Implementation Notes

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

LDDQU (128-bit Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

VLDDQU (VEX.128 encoded version) $\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$ $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$ **VLDDQU (VEX.256 encoded version)** $\text{DEST}[255:0] \leftarrow \text{SRC}[255:0]$ **Intel C/C++ Compiler Intrinsic Equivalent**LDDQU: `__m128i _mm_lddqu_si128 (__m128i * p);`VLDDQU: `__m256i _mm256_lddqu_si256 (__m256i * p);`**Numeric Exceptions**

None.

Other Exceptions

See Exceptions Type 4;

Note treatment of #AC varies.

LDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F,AE,12 LDMXCSR <i>m32</i>	M	V/V	SSE	Load MXCSR register from <i>m32</i> .
VEX.LZ.0F.WIG AE 12 VLDMXCSR <i>m32</i>	M	V/V	AVX	Load MXCSR register from <i>m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See “MXCSR Control and Status Register” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

$\text{MXCSR} \leftarrow \text{m32};$

C/C++ Compiler Intrinsic Equivalent

`_mm_setcsr(unsigned int i)`

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

- #GP For an attempt to set reserved bits in MXCSR.
- #UD If VEX.vvvv ≠ 1111B.

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C5 /r	LDS <i>r16,m16:16</i>	RM	Invalid	Valid	Load DS: <i>r16</i> with far pointer from memory.
C5 /r	LDS <i>r32,m16:32</i>	RM	Invalid	Valid	Load DS: <i>r32</i> with far pointer from memory.
OF B2 /r	LSS <i>r16,m16:16</i>	RM	Valid	Valid	Load SS: <i>r16</i> with far pointer from memory.
OF B2 /r	LSS <i>r32,m16:32</i>	RM	Valid	Valid	Load SS: <i>r32</i> with far pointer from memory.
REX + OF B2 /r	LSS <i>r64,m16:64</i>	RM	Valid	N.E.	Load SS: <i>r64</i> with far pointer from memory.
C4 /r	LES <i>r16,m16:16</i>	RM	Invalid	Valid	Load ES: <i>r16</i> with far pointer from memory.
C4 /r	LES <i>r32,m16:32</i>	RM	Invalid	Valid	Load ES: <i>r32</i> with far pointer from memory.
OF B4 /r	LFS <i>r16,m16:16</i>	RM	Valid	Valid	Load FS: <i>r16</i> with far pointer from memory.
OF B4 /r	LFS <i>r32,m16:32</i>	RM	Valid	Valid	Load FS: <i>r32</i> with far pointer from memory.
REX + OF B4 /r	LFS <i>r64,m16:64</i>	RM	Valid	N.E.	Load FS: <i>r64</i> with far pointer from memory.
OF B5 /r	LGS <i>r16,m16:16</i>	RM	Valid	Valid	Load GS: <i>r16</i> with far pointer from memory.
OF B5 /r	LGS <i>r32,m16:32</i>	RM	Valid	Valid	Load GS: <i>r32</i> with far pointer from memory.
REX + OF B5 /r	LGS <i>r64,m16:64</i>	RM	Valid	N.E.	Load GS: <i>r64</i> with far pointer from memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

Operation**64-BIT_MODE**

```

IF SS is loaded
  THEN
    IF SegmentSelector = NULL and ( (RPL = 3) or
      (RPL ≠ 3 and RPL ≠ CPL) )
      THEN #GP(0);
    ELSE IF descriptor is in non-canonical space
      THEN #GP(0); FI;
    ELSE IF Segment selector index is not within descriptor table limits
      or segment selector RPL ≠ CPL
      or access rights indicate nonwritable data segment
      or DPL ≠ CPL
      THEN #GP(selector); FI;
    ELSE IF Segment marked not present
      THEN #SS(selector); FI;
    FI;
    SS ← SegmentSelector(SRC);
    SS ← SegmentDescriptor([SRC]);
  ELSE IF attempt to load DS, or ES
    THEN #UD;
  ELSE IF FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
      or access rights indicate segment neither data nor readable code segment
      or segment is data or nonconforming-code segment
      and ( RPL > DPL or CPL > DPL )
      THEN #GP(selector); FI;
    ELSE IF Segment marked not present
      THEN #NP(selector); FI;
    FI;
    SegmentRegister ← SegmentSelector(SRC);
    SegmentRegister ← SegmentDescriptor([SRC]);
  FI;
  ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
      SegmentRegister ← NULLSelector;
      SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag;
        not accessible by software *)
  FI;
  DEST ← Offset(SRC);

```

PROTECTED MODE OR COMPATIBILITY MODE;

```

IF SS is loaded
  THEN
    IF SegmentSelector = NULL
      THEN #GP(0);
    ELSE IF Segment selector index is not within descriptor table limits
      or segment selector RPL ≠ CPL
      or access rights indicate nonwritable data segment
      or DPL ≠ CPL
      THEN #GP(selector); FI;
    ELSE IF Segment marked not present
      THEN #SS(selector); FI;
    FI;

```

```

    SS ← SegmentSelector(SRC);
    SS ← SegmentDescriptor([SRC]);
ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and (RPL > DPL or CPL > DPL)
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #NP(selector); FI;
        FI;
    SegmentRegister ← SegmentSelector(SRC) AND RPL;
    SegmentRegister ← SegmentDescriptor([SRC]);
FI;
ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister ← NULLSelector;
        SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag;
            not accessible by software *)
FI;
DEST ← Offset(SRC);

Real-Address or Virtual-8086 Mode
    SegmentRegister ← SegmentSelector(SRC); FI;
    DEST ← Offset(SRC);

```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a NULL selector is loaded into the SS register. 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.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL. If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#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.
#UD	If source operand is not a memory location. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a NULL selector is attempted to be loaded into the SS register in compatibility mode. If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode. If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.
#GP(Selector)	If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL. If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.
#SS(0)	If a memory operand effective address is non-canonical
#SS(Selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#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.
#UD	If source operand is not a memory location. If the LOCK prefix is used.

LEA—Load Effective Address

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
8D /r	LEA <i>r16,m</i>	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r16</i> .
8D /r	LEA <i>r32,m</i>	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r32</i> .
REX.W + 8D /r	LEA <i>r64,m</i>	RM	Valid	N.E.	Store effective address for <i>m</i> in register <i>r64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

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

Table 3-62. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes

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

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

In 64-bit mode, the instruction's destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-63.

Table 3-63. 64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	32	32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
16	64	64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
32	32	32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.
32	64	64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.
64	32	32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).
64	64	64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).

Operation

IF OperandSize = 16 and AddressSize = 16

THEN

DEST \leftarrow EffectiveAddress(SRC); (* 16-bit address *)

ELSE IF OperandSize = 16 and AddressSize = 32

THEN

temp \leftarrow EffectiveAddress(SRC); (* 32-bit address *)

DEST \leftarrow temp[0:15]; (* 16-bit address *)

FI;

ELSE IF OperandSize = 32 and AddressSize = 16

THEN

temp \leftarrow EffectiveAddress(SRC); (* 16-bit address *)

DEST \leftarrow ZeroExtend(temp); (* 32-bit address *)

FI;

ELSE IF OperandSize = 32 and AddressSize = 32

THEN

DEST \leftarrow EffectiveAddress(SRC); (* 32-bit address *)

FI;

ELSE IF OperandSize = 16 and AddressSize = 64

THEN

temp \leftarrow EffectiveAddress(SRC); (* 64-bit address *)

DEST \leftarrow temp[0:15]; (* 16-bit address *)

FI;

ELSE IF OperandSize = 32 and AddressSize = 64

THEN

temp \leftarrow EffectiveAddress(SRC); (* 64-bit address *)

DEST \leftarrow temp[0:31]; (* 16-bit address *)

FI;

ELSE IF OperandSize = 64 and AddressSize = 64

THEN

DEST \leftarrow EffectiveAddress(SRC); (* 64-bit address *)

FI;

FI;

Flags Affected

None.

Protected Mode Exceptions

#UD If source operand is not a memory location.
If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C9	LEAVE	NP	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	NP	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	NP	Valid	N.E.	Set RSP to RBP, then pop RBP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

See "Procedure Calls for Block-Structured Languages" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF StackAddressSize = 32
  THEN
    ESP ← EBP;
  ELSE IF StackAddressSize = 64
    THEN RSP ← RBP; FI;
  ELSE IF StackAddressSize = 16
    THEN SP ← BP; FI;
FI;
```

```
IF OperandSize = 32
  THEN EBP ← Pop();
  ELSE IF OperandSize = 64
    THEN RBP ← Pop(); FI;
  ELSE IF OperandSize = 16
    THEN BP ← Pop(); FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0) If the EBP register points to a location that is not within the limits of the current stack segment.

#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LFENCE—Load Fence

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AE E8	LFENCE	NP	Valid	Valid	Serializes load operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete **before** the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

Operation

```
Wait_On_Following_Instructions_Until(preceding_instructions_complete);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_lfence(void)
```

Exceptions (All Modes of Operation)

#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
If the LOCK prefix is used.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /2	LGDT <i>m16&32</i>	M	N.E.	Valid	Load <i>m</i> into GDTR.
OF 01 /3	LIDT <i>m16&32</i>	M	N.E.	Valid	Load <i>m</i> into IDTR.
OF 01 /2	LGDT <i>m16&64</i>	M	Valid	N.E.	Load <i>m</i> into GDTR.
OF 01 /3	LIDT <i>m16&64</i>	M	Valid	N.E.	Load <i>m</i> into IDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

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

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

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See “SGDT—Store Global Descriptor Table Register” in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.

Operation

IF Instruction is LIDT

THEN

IF OperandSize = 16

THEN

IDTR(Limit) ← SRC[0:15];

IDTR(Base) ← SRC[16:47] AND 00FFFFFFFH;

ELSE IF 32-bit Operand Size

THEN

IDTR(Limit) ← SRC[0:15];

IDTR(Base) ← SRC[16:47];

FI;

ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)

THEN

IDTR(Limit) ← SRC[0:15];

IDTR(Base) ← SRC[16:79];

FI;

FI;

ELSE (* Instruction is LGDT *)

IF OperandSize = 16

```

    THEN
        GDTR(Limit) ← SRC[0:15];
        GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
    ELSE IF 32-bit Operand Size
        THEN
            GDTR(Limit) ← SRC[0:15];
            GDTR(Base) ← SRC[16:47];
        FI;
    ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
        THEN
            GDTR(Limit) ← SRC[0:15];
            GDTR(Base) ← SRC[16:79];
        FI;
    FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If the current privilege level is not 0. 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.

Real-Address Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#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.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	The LGDT and LIDT instructions are not recognized in virtual-8086 mode.
#GP	If the current privilege level is not 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0. If the memory address is in a non-canonical form.
#UD	If source operand is not a memory location. If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /2	LLDT <i>r/m16</i>	M	Valid	Valid	Load segment selector <i>r/m16</i> into LDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA	NA

Description

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

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

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

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

Operation

IF SRC(Offset) > descriptor table limit
THEN #GP(segment selector); FI;

IF segment selector is valid

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ LDT
THEN #GP(segment selector); FI;

IF segment descriptor is not present
THEN #NP(segment selector); FI;

LDTR(SegmentSelector) ← SRC;

LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;

ELSE LDTR ← INVALID

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. 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.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table. Segment selector is beyond GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LLDT instruction is not recognized in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

#UD	The LLDT instruction is not recognized in virtual-8086 mode.
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0. If the memory address is in a non-canonical form.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table. Segment selector is beyond GDT limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LMSW—Load Machine Status Word

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /6	LMSW <i>r/m16</i>	M	Valid	Valid	Loads <i>r/m16</i> in machine status word of CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA	NA

Description

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

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

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

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

CR0[0:3] ← SRC[0:3];

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. 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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	The LMSW instruction is not recognized in real-address mode.
--------	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0. If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FO	LOCK	NP	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

Note that, in later Intel 64 and IA-32 processors (including the Pentium 4, Intel Xeon, and P6 family processors), locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

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

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

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on locking of caches.

Operation

AssertLOCK#(DurationOfAccompanyingInstruction);

Flags Affected

None.

Protected Mode Exceptions

#UD If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.
Other exceptions can be generated by the instruction when the LOCK prefix is applied.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

LODS/LODSB/LODSW/LODSD/LODSQ—Load String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
AC	LODS <i>m8</i>	NP	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODS <i>m16</i>	NP	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODS <i>m32</i>	NP	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODS <i>m64</i>	NP	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	NP	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODSW	NP	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODSD	NP	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODSQ	NP	Valid	N.E.	Load qword at address (R)SI into RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

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

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

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See “REP/REPE/REPZ /REPNE/REPZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

Operation

```

IF AL ← SRC; (* Byte load *)
    THEN AL ← SRC; (* Byte load *)
        IF DF = 0
            THEN (E)SI ← (E)SI + 1;
            ELSE (E)SI ← (E)SI - 1;
        FI;
ELSE IF AX ← SRC; (* Word load *)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 2;
        ELSE (E)SI ← (E)SI - 2;
    IF;
    FI;
ELSE IF EAX ← SRC; (* Doubleword load *)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 4;
        ELSE (E)SI ← (E)SI - 4;
    FI;
    FI;
ELSE IF RAX ← SRC; (* Quadword load *)
    THEN IF DF = 0
        THEN (R)SI ← (R)SI + 8;
        ELSE (R)SI ← (R)SI - 8;
    FI;
    FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

LOOP/LOOP_{cc}—Loop According to ECX Counter

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E2 <i>cb</i>	LOOP <i>rel8</i>	D	Valid	Valid	Decrement count; jump short if count \neq 0.
E1 <i>cb</i>	LOOPE <i>rel8</i>	D	Valid	Valid	Decrement count; jump short if count \neq 0 and ZF = 1.
E0 <i>cb</i>	LOOPNE <i>rel8</i>	D	Valid	Valid	Decrement count; jump short if count \neq 0 and ZF = 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

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

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

Operation

```

IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;

Count ← Count - 1;

IF Instruction is not LOOP
    THEN
        IF (Instruction ← LOOPE) or (Instruction ← LOOPZ)
            THEN IF (ZF = 1) and (Count  $\neq$  0)
                THEN BranchCond ← 1;
                ELSE BranchCond ← 0;
            FI;
        ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
            IF (ZF = 0) and (Count  $\neq$  0)
                THEN BranchCond ← 1;
                ELSE BranchCond ← 0;

```



```

        FI;
    FI;
ELSE (* Instruction = LOOP *)
    IF (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
    FI;
FI;

IF BranchCond = 1
    THEN
        IF OperandSize = 32
            THEN EIP ← EIP + SignExtend(DEST);
            ELSE IF OperandSize = 64
                THEN RIP ← RIP + SignExtend(DEST);
                FI;
            ELSE IF OperandSize = 16
                THEN EIP ← EIP AND 0000FFFFH;
                FI;
        FI;
        IF OperandSize = (32 or 64)
            THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit
                #GP; FI;
            FI;
        FI;
    ELSE
        Terminate loop and continue program execution at (R/E)IP;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

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

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.
 #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the offset being jumped to is in a non-canonical form.
 #UD If the LOCK prefix is used.

LSL—Load Segment Limit

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 03 /r	LSL <i>r16</i> , <i>r16/m16</i>	RM	Valid	Valid	Load: <i>r16</i> ← segment limit, selector <i>r16/m16</i> .
OF 03 /r	LSL <i>r32</i> , <i>r32/m16</i> *	RM	Valid	Valid	Load: <i>r32</i> ← segment limit, selector <i>r32/m16</i> .
REX.W + OF 03 /r	LSL <i>r64</i> , <i>r32/m16</i> *	RM	Valid	Valid	Load: <i>r64</i> ← segment limit, selector <i>r32/m16</i> .

NOTES:

* For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

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

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

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

Table 3-64. Segment and Gate Descriptor Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Upper 8 byte of a 16-Byte descriptor	Yes
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	No	Reserved	No
5	16-bit/32-bit task gate	No	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	No	64-bit call gate	No
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

IF SRC(Offset) > descriptor table limit
 THEN ZF ← 0; FI;

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment
 and (CPL > DPL) OR (RPL > DPL)
 or Segment type is not valid for instruction
 THEN
 ZF ← 0;
 ELSE
 temp ← SegmentLimit([SRC]);
 IF (G ← 1)
 THEN temp ← ShiftLeft(12, temp) OR 00000FFFH;
 ELSE IF OperandSize = 32
 THEN DEST ← temp; FI;
 ELSE IF OperandSize = 64 (* REX.W used *)
 THEN DEST (* Zero-extended *) ← temp; FI;
 ELSE (* OperandSize = 16 *)
 DEST ← temp AND FFFFH;
 FI;

FI;

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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 the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LTR—Load Task Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /3	LTR <i>r/m16</i>	M	Valid	Valid	Load <i>r/m16</i> into task register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA	NA

Description

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

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

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

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

Operation

IF SRC is a NULL selector
THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
THEN #GP(segment selector); FI;

Read segment descriptor;

IF segment descriptor is not for an available TSS
THEN #GP(segment selector); FI;

IF segment descriptor is not present
THEN #NP(segment selector); FI;

TSSsegmentDescriptor(busy) ← 1;

(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)

TaskRegister(SegmentSelector) ← SRC;

TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the source operand contains a NULL segment selector.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If the source selector points to a segment that is not a TSS or to one for a task that is already busy.</p> <p>If the selector points to LDT or is beyond the GDT limit.</p>
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LTR instruction is not recognized in real-address mode.
-----	---

Virtual-8086 Mode Exceptions

#UD	The LTR instruction is not recognized in virtual-8086 mode.
-----	---

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory address is in a non-canonical form.</p> <p>If the source operand contains a NULL segment selector.</p>
#GP(selector)	<p>If the source selector points to a segment that is not a TSS or to one for a task that is already busy.</p> <p>If the selector points to LDT or is beyond the GDT limit.</p> <p>If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.</p>
#NP(selector)	If the TSS is marked not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LZCNT— Count the Number of Leading Zero Bits

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BD /r LZCNT r16, r/m16	RM	V/V	LZCNT	Count the number of leading zero bits in r/m16, return result in r16.
F3 0F BD /r LZCNT r32, r/m32	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
REX.W + F3 0F BD /r LZCNT r64, r/m64	RM	V/N.E.	LZCNT	Count the number of leading zero bits in r/m64, return result in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR.

In 64-bit mode 64-bit operand size requires REX.W=1.

Operation

```

temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp ← temp - 1
    DEST ← DEST + 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI

IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI

```

Flags Affected

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

LZCNT: unsigned __int32 _lzcnt_u32(unsigned __int32 src);

LZCNT: unsigned __int64 _lzcnt_u64(unsigned __int64 src);

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#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(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU <i>xmm1</i> , <i>xmm2</i>	RM	V/V	SSE2	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU <i>xmm1</i> , <i>xmm2</i>	RM	V/V	AVX	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If VMASKMOVDQU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

¹.ModRM.MOD = 011B required

Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI + 1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DI/EDI + 15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)
```

Other Exceptions

See Exceptions Type 4; additionally

```

#UD                If VEX.L = 1
                    If VEX.vvvv ≠ 1111B.

```

MASKMOVQ—Store Selected Bytes of Quadword

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF F7 /r MASKMOVQ <i>mm1</i> , <i>mm2</i>	RM	Valid	Valid	Selectively write bytes from <i>mm1</i> to memory location using the byte mask in <i>mm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI + 1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI + 15] ← SRC[63:56] ELSE (* Memory location unchanged *); FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmove_si64(__m64d, __m64n, char * p)
```

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

MAXPD—Return Maximum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5F /r MAXPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Return the maximum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the maximum double-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the maximum packed double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MAX(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
  FI;
}
```

MAXPD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{MAX}(\text{DEST}[127:64], \text{SRC}[127:64])$$

$$\text{DEST}[\text{VLMAX}-1:128] \text{ (Unmodified)}$$
VMAXPD (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VMAXPD (VEX.256 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])$$

$$\text{DEST}[191:128] \leftarrow \text{MAX}(\text{SRC1}[191:128], \text{SRC2}[191:128])$$

$$\text{DEST}[255:192] \leftarrow \text{MAX}(\text{SRC1}[255:192], \text{SRC2}[255:192])$$
Intel C/C++ Compiler Intrinsic Equivalent

MAXPD: `__m128d _mm_max_pd(__m128d a, __m128d b);`

VMAXPD: `__m256d _mm256_max_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MAXPS—Return Maximum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5F /r MAXPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Return the maximum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.OF.WIG 5F /r VMAXPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the maximum single-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 5F /r VMAXPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the maximum single double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MAX(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```

MAXPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{DEST}[31:0], \text{SRC}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{DEST}[63:32], \text{SRC}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{DEST}[95:64], \text{SRC}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{DEST}[127:96], \text{SRC}[127:96])$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VMAXPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96])$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMAXPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96])$
 $\text{DEST}[159:128] \leftarrow \text{MAX}(\text{SRC1}[159:128], \text{SRC2}[159:128])$
 $\text{DEST}[191:160] \leftarrow \text{MAX}(\text{SRC1}[191:160], \text{SRC2}[191:160])$
 $\text{DEST}[223:192] \leftarrow \text{MAX}(\text{SRC1}[223:192], \text{SRC2}[223:192])$
 $\text{DEST}[255:224] \leftarrow \text{MAX}(\text{SRC1}[255:224], \text{SRC2}[255:224])$

Intel C/C++ Compiler Intrinsic Equivalent

MAXPS: `__m128 _mm_max_ps (__m128 a, __m128 b);`
 VMAXPS: `__m256 _mm256_max_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD <i>xmm1</i> , <i>xmm2/mem64</i>	RM	V/V	SSE2	Return the maximum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/mem64</i>	RVM	V/V	AVX	Return the maximum scalar double-precision floating-point value between <i>xmm3/mem64</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and second the source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed. The high quadword of the destination operand is copied from the same bits of first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MAX(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
  FI;
}
```

MAXSD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$$

$$\text{DEST}[\text{VLMAX}-1:64] \text{ (Unmodified)}$$
VMAXSD (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
Intel C/C++ Compiler Intrinsic Equivalent

MAXSD: `__m128d _mm_max_sd(__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS <i>xmm1</i> , <i>xmm2/mem32</i>	RM	V/V	SSE	Return the maximum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/mem32</i>	RVM	V/V	AVX	Return the maximum scalar single-precision floating-point value between <i>xmm3/mem32</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MAX(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```

MAXSS (128-bit Legacy SSE version)

```
DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[VLMAX-1:32] (Unmodified)
```

VMAXSS (VEX.128 encoded version) $\text{DEST}[31:0] \leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$ $\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$ $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$ **Intel C/C++ Compiler Intrinsic Equivalent**`__m128d _mm_max_ss(__m128d a, __m128d b)`**SIMD Floating-Point Exceptions**

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MFENCE—Memory Fence

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F AE F0	MFENCE	NP	Valid	Valid	Serializes load and store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.¹ The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

Operation

Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_mfence(void)

Exceptions (All Modes of Operation)

#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
If the LOCK prefix is used.

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

MINPD—Return Minimum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5D /r MINPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Return the minimum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5D /r VMINPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the minimum double-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5D /r VMINPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the minimum packed double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

MIN(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```

MINPD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$$

$$\text{DEST}[\text{VLMAX}-1:128] \text{ (Unmodified)}$$
VMINPD (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VMINPD (VEX.256 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$$

$$\text{DEST}[191:128] \leftarrow \text{MIN}(\text{SRC1}[191:128], \text{SRC2}[191:128])$$

$$\text{DEST}[255:192] \leftarrow \text{MIN}(\text{SRC1}[255:192], \text{SRC2}[255:192])$$
Intel C/C++ Compiler Intrinsic Equivalent

MINPD: `__m128d _mm_min_pd(__m128d a, __m128d b);`

VMINPD: `__m256d _mm256_min_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MINPS—Return Minimum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5D /r MINPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Return the minimum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.OF.WIG 5D /r VMINPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the minimum single-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 5D /r VMINPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the minimum single double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MIN(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```


MINPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96])$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VMINPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96])$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMINPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96])$
 $\text{DEST}[159:128] \leftarrow \text{MIN}(\text{SRC1}[159:128], \text{SRC2}[159:128])$
 $\text{DEST}[191:160] \leftarrow \text{MIN}(\text{SRC1}[191:160], \text{SRC2}[191:160])$
 $\text{DEST}[223:192] \leftarrow \text{MIN}(\text{SRC1}[223:192], \text{SRC2}[223:192])$
 $\text{DEST}[255:224] \leftarrow \text{MIN}(\text{SRC1}[255:224], \text{SRC2}[255:224])$

Intel C/C++ Compiler Intrinsic Equivalent

MINPS: `__m128d _mm_min_ps(__m128d a, __m128d b);`
 VMINPS: `__m256 _mm256_min_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5D /r MINSD <i>xmm1</i> , <i>xmm2/mem64</i>	RM	V/V	SSE2	Return the minimum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/mem64</i>	RVM	V/V	AVX	Return the minimum scalar double precision floating-point value between <i>xmm3/mem64</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed. The high quadword of the destination operand is copied from the same bits in the first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MIN(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
      ELSE DEST ← SRC2;
    FI;
}
```

MINSD (128-bit Legacy SSE version)

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[VLMAX-1:64] (Unmodified)

MINSD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$

$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MINSD: `__m128d _mm_min_sd(__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 5D /r MINSS <i>xmm1, xmm2/m32</i>	RM	V/V	SSE	Return the minimum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5D /r VMINSS <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Return the minimum scalar single precision floating-point value between <i>xmm3/mem32</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MIN(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```

MINSS (128-bit Legacy SSE version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[VLMAX-1:32] (Unmodified)
```

VMINSS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$

$\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MINSS: `__m128d _mm_min_ss(__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 C8	MONITOR	NP	Valid	Valid	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:EAX (DS:RAX in 64-bit mode).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The content of EAX is an effective address (in 64-bit mode, RAX is used). By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, “Multiple-Processor Management” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: `void _mm_monitor(void const *p, unsigned extensions,unsigned hints)`

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If the value in EAX 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. If ECX \neq 0.
#SS(0)	If the value in EAX is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0. If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH. If ECX \neq 0.
#SS	If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form. If RCX \neq 0.
#SS(0)	If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If the current privilege level is not 0. If CPUID.01H:ECX.MONITOR[bit 3] = 0.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8 ^{***} ,r8 ^{***}	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64,r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8 ^{***} ,r/m8 ^{***}	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16,Sreg ^{**}	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r/m64,Sreg ^{**}	MR	Valid	Valid	Move zero extended 16-bit segment register to r/m64.
8E /r	MOV Sreg,r/m16 ^{**}	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64 ^{**}	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8*	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8*	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16*	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64*	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8,AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8 ^{***} ,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16*,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64*,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8,imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8 ^{***} ,imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16,imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32,imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64,imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /O ib	MOV r/m8,imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /O ib	MOV r/m8 ^{***} ,imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /O iw	MOV r/m16,imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /O id	MOV r/m32,imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /O io	MOV r/m64,imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

NOTES:

- * The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 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).
- ***In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

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, a doubleword, or a quadword.

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

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

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

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an interrupt occurs¹. Be aware that 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

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that load the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before MOV ESP, EBP executes:

```
MOV SS, EDX
MOV SS, EAX
MOV ESP, EBP
```

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 4, Intel Xeon, and P6 family processors, the two high-order bytes are filled with zeros; for earlier 32-bit IA-32 processors, the two high order bytes are undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST \leftarrow 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 to which it points.

```
IF SS is loaded
  THEN
    IF segment selector is NULL
      THEN #GP(0); FI;
    IF segment selector index is outside descriptor table limits
      or segment selector's RPL  $\neq$  CPL
      or segment is not a writable data segment
      or DPL  $\neq$  CPL
      THEN #GP(selector); FI;
    IF segment not marked present
      THEN #SS(selector);
    ELSE
      SS  $\leftarrow$  segment selector;
      SS  $\leftarrow$  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)
      or ((RPL > DPL) and (CPL > DPL))
      THEN #GP(selector); FI;
    IF segment not marked present
      THEN #NP(selector);
    ELSE
      SegmentRegister  $\leftarrow$  segment selector;
      SegmentRegister  $\leftarrow$  segment descriptor; FI;
```

FI;

```
IF DS, ES, FS, or GS is loaded with NULL selector
  THEN
    SegmentRegister  $\leftarrow$  segment selector;
    SegmentRegister  $\leftarrow$  segment descriptor;
```

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If attempt is made to load SS register with NULL segment selector.</p> <p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p> <p>If the SS register is being loaded and the segment pointed to is a non-writable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#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.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

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.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	<p>If the memory address is in a non-canonical form.</p> <p>If an attempt is made to load SS register with NULL segment selector when CPL = 3.</p> <p>If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the memory access to the descriptor table is non-canonical.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p> <p>If the SS register is being loaded and the segment pointed to is a nonwritable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.</p>
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#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.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 20/r MOV r32, CR0–CR7	MR	N.E.	Valid	Move control register to r32.
OF 20/r MOV r64, CR0–CR7	MR	Valid	N.E.	Move extended control register to r64.
REX.R + OF 20 /0 MOV r64, CR8	MR	Valid	N.E.	Move extended CR8 to r64. ¹
OF 22 /r MOV CR0–CR7, r32	RM	N.E.	Valid	Move r32 to control register.
OF 22 /r MOV CR0–CR7, r64	RM	Valid	N.E.	Move r64 to extended control register.
REX.R + OF 22 /0 MOV CR8, r64	RM	Valid	N.E.	Move r64 to extended CR8. ¹

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8–R15). Use of the REX.W prefix or 66H prefix is ignored. Use of

the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

DEST ← SRC;

Flags Affected

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

Protected Mode Exceptions

- #GP(0)
 - If the current privilege level is not 0.
 - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
 - If an attempt is made to write a 1 to any reserved bit in CR4.
 - If an attempt is made to write 1 to CR4.PCIDE.
 - If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
- #UD
 - If the LOCK prefix is used.
 - If an attempt is made to access CR1, CR5, CR6, or CR7.

Real-Address Mode Exceptions

- #GP
 - If an attempt is made to write a 1 to any reserved bit in CR4.
 - If an attempt is made to write 1 to CR4.PCIDE.
 - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
- #UD
 - If the LOCK prefix is used.
 - If an attempt is made to access CR1, CR5, CR6, or CR7.

Virtual-8086 Mode Exceptions

- #GP(0)
 - These instructions cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

- #GP(0)
 - If the current privilege level is not 0.
 - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
 - If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
 - If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
 - If an attempt is made to write a 1 to any reserved bit in CR3.
 - If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
- #UD
 - If the LOCK prefix is used.
 - If an attempt is made to access CR1, CR5, CR6, or CR7.

64-Bit Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).</p> <p>If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.</p> <p>If an attempt is made to clear CR0.PG[bit 31].</p> <p>If an attempt is made to write a 1 to any reserved bit in CR4.</p> <p>If an attempt is made to write a 1 to any reserved bit in CR8.</p> <p>If an attempt is made to write a 1 to any reserved bit in CR3.</p> <p>If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, or CR7.</p> <p>If the REX.R prefix is used to specify a register other than CR8.</p>

MOV—Move to/from Debug Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 21/ <i>r</i> MOV <i>r32</i> , DR0-DR7	MR	N.E.	Valid	Move debug register to <i>r32</i> .
OF 21/ <i>r</i> MOV <i>r64</i> , DR0-DR7	MR	Valid	N.E.	Move extended debug register to <i>r64</i> .
OF 23 / <i>r</i> MOV DR0-DR7, <i>r32</i>	RM	N.E.	Valid	Move <i>r32</i> to debug register.
OF 23 / <i>r</i> MOV DR0-DR7, <i>r64</i>	RM	Valid	N.E.	Move <i>r64</i> to extended debug register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 17.2, “Debug Registers”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the flags and fields in the debug registers.)

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

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

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

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8–R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ← SRC;
```

FI;

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

Real-Address Mode Exceptions

#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

Virtual-8086 Mode Exceptions

#GP(0)	The debug registers cannot be loaded or read when in virtual-8086 mode.
--------	---

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0. If an attempt is made to write a 1 to any of bits 63:32 in DR6. If an attempt is made to write a 1 to any of bits 63:32 in DR7.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. If the LOCK prefix is used. If the REX.R prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 29 /r MOVAPD <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 28 /r VMOVAPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Move aligned packed double-precision floating-point values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 29 /r VMOVAPD <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move aligned packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 28 /r VMOVAPD <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move aligned packed double-precision floating-point values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.66.0F.WIG 29 /r VMOVAPD <i>ymm2/m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move aligned packed double-precision floating-point values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 2 or 4 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated.

To move double-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPD instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit versions: Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

(V)MOVAPD (128-bit store-form version)

DEST[127:0] ← SRC[127:0]

VMOVAPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] ← 0

VMOVAPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVAPD: __m128d _mm_load_pd (double const * p);

MOVAPD: _mm_store_pd(double * p, __m128d a);

VMOVAPD: __m256d _mm256_load_pd (double const * p);

VMOVAPD: _mm256_store_pd(double * p, __m256d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 28 /r MOVAPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
OF 29 /r MOVAPS <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.OF.WIG 28 /r VMOVAPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Move aligned packed single-precision floating-point values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.OF.WIG 29 /r VMOVAPS <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move aligned packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.OF.WIG 28 /r VMOVAPS <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move aligned packed single-precision floating-point values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.OF.WIG 29 /r VMOVAPS <i>ymm2/m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move aligned packed single-precision floating-point values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 4 or 8 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from a 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated.

To move single-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPS instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Operation

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] (Unmodified)

(V)MOVAPS (128-bit store form)

DEST[127:0] ← SRC[127:0]

VMOVAPS (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVAPS (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVAPS: __m128 _mm_load_ps (float const * p);
MOVAPS: _mm_store_ps(float * p, __m128 a);
VMOVAPS: __m256 _mm256_load_ps (float const * p);
VMOVAPS: _mm256_store_ps(float * p, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE; additionally
#UD If VEX.vvvv ≠ 1111B.

MOVBE—Move Data After Swapping Bytes

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 38 F0 /r	MOVBE <i>r16</i> , <i>m16</i>	RM	Valid	Valid	Reverse byte order in <i>m16</i> and move to <i>r16</i> .
OF 38 F0 /r	MOVBE <i>r32</i> , <i>m32</i>	RM	Valid	Valid	Reverse byte order in <i>m32</i> and move to <i>r32</i> .
REX.W + OF 38 F0 /r	MOVBE <i>r64</i> , <i>m64</i>	RM	Valid	N.E.	Reverse byte order in <i>m64</i> and move to <i>r64</i> .
OF 38 F1 /r	MOVBE <i>m16</i> , <i>r16</i>	MR	Valid	Valid	Reverse byte order in <i>r16</i> and move to <i>m16</i> .
OF 38 F1 /r	MOVBE <i>m32</i> , <i>r32</i>	MR	Valid	Valid	Reverse byte order in <i>r32</i> and move to <i>m32</i> .
REX.W + OF 38 F1 /r	MOVBE <i>m64</i> , <i>r64</i>	MR	Valid	N.E.	Reverse byte order in <i>r64</i> and move to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

TEMP ← SRC

IF (OperandSize = 16)

THEN

DEST[7:0] ← TEMP[15:8];

DEST[15:8] ← TEMP[7:0];

ELSE IF (OperandSize = 32)

DEST[7:0] ← TEMP[31:24];

DEST[15:8] ← TEMP[23:16];

DEST[23:16] ← TEMP[15:8];

DEST[31:23] ← TEMP[7:0];

ELSE IF (OperandSize = 64)

DEST[7:0] ← TEMP[63:56];

DEST[15:8] ← TEMP[55:48];

DEST[23:16] ← TEMP[47:40];

DEST[31:24] ← TEMP[39:32];

DEST[39:32] ← TEMP[31:24];

DEST[47:40] ← TEMP[23:16];

DEST[55:48] ← TEMP[15:8];

DEST[63:56] ← TEMP[7:0];

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination operand is in a non-writable 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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used. If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 6E /r MOVD <i>mm</i> , <i>r/m32</i>	RM	V/V	MMX	Move doubleword from <i>r/m32</i> to <i>mm</i> .
REX.W + 0F 6E /r MOVQ <i>mm</i> , <i>r/m64</i>	RM	V/N.E.	MMX	Move quadword from <i>r/m64</i> to <i>mm</i> .
0F 7E /r MOVD <i>r/m32</i> , <i>mm</i>	MR	V/V	MMX	Move doubleword from <i>mm</i> to <i>r/m32</i> .
REX.W + 0F 7E /r MOVQ <i>r/m64</i> , <i>mm</i>	MR	V/N.E.	MMX	Move quadword from <i>mm</i> to <i>r/m64</i> .
VEX.128.66.0F.W0 6E / VMOVD <i>xmm1</i> , <i>r32/m32</i>	RM	V/V	AVX	Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
VEX.128.66.0F.W1 6E /r VMOVQ <i>xmm1</i> , <i>r64/m64</i>	RM	V/N.E.	AVX	Move quadword from <i>r/m64</i> to <i>xmm1</i> .
66 0F 6E /r MOVD <i>xmm</i> , <i>r/m32</i>	RM	V/V	SSE2	Move doubleword from <i>r/m32</i> to <i>xmm</i> .
66 REX.W 0F 6E /r MOVQ <i>xmm</i> , <i>r/m64</i>	RM	V/N.E.	SSE2	Move quadword from <i>r/m64</i> to <i>xmm</i> .
66 0F 7E /r MOVD <i>r/m32</i> , <i>xmm</i>	MR	V/V	SSE2	Move doubleword from <i>xmm</i> register to <i>r/m32</i> .
66 REX.W 0F 7E /r MOVQ <i>r/m64</i> , <i>xmm</i>	MR	V/N.E.	SSE2	Move quadword from <i>xmm</i> register to <i>r/m64</i> .
VEX.128.66.0F.W0 7E /r VMOVD <i>r32/m32</i> , <i>xmm1</i>	MR	V/V	AVX	Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
VEX.128.66.0F.W1 7E /r VMOVQ <i>r64/m64</i> , <i>xmm1</i>	MR	V/N.E.	AVX	Move quadword from <i>xmm1</i> register to <i>r/m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

MOVD (when destination operand is MMX technology register)

DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;

MOVD (when destination operand is XMM register)

DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000H;
DEST[VLMAX-1:128] (Unmodified)

MOVD (when source operand is MMX technology or XMM register)

DEST ← SRC[31:0];

VMOVD (VEX-encoded version when destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[VLMAX-1:32] ← 0

MOVQ (when destination operand is XMM register)

DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[VLMAX-1:128] (Unmodified)

MOVQ (when destination operand is r/m64)

DEST[63:0] ← SRC[63:0];

MOVQ (when source operand is XMM register or r/m64)

DEST ← SRC[63:0];

VMOVQ (VEX-encoded version when destination is an XMM register)

DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVD: __m64 _mm_cvtsi32_si64 (int i)
MOVD: int _mm_cvtsi64_si32 (__m64m)
MOVD: __m128i _mm_cvtsi32_si128 (int a)
MOVD: int _mm_cvtsi128_si32 (__m128i a)
MOVQ: __int64 _mm_cvtsi128_si64(__m128i);
MOVQ: __m128i _mm_cvtsi64_si128(__int64);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

MOVDDUP—Move One Double-FP and Duplicate

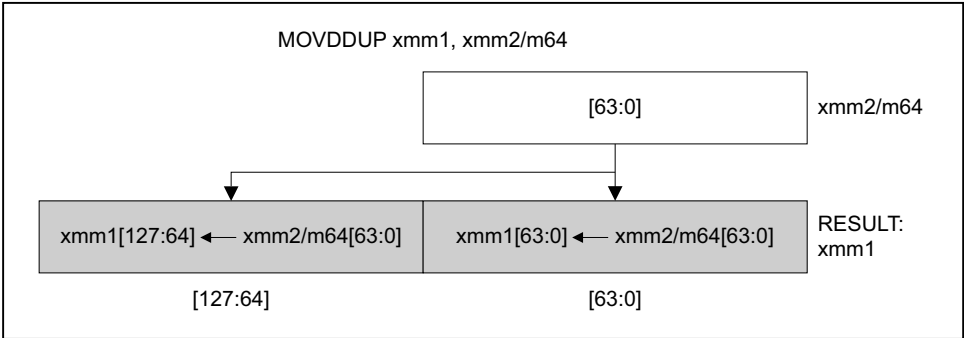
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE3	Move one double-precision floating-point value from the lower 64-bit operand in <i>xmm2/m64</i> to <i>xmm1</i> and duplicate.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Move double-precision floating-point values from <i>xmm2/mem</i> and duplicate into <i>xmm1</i> .
VEX.256.F2.0F.WIG 12 /r VMOVDDUP <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move even index double-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 8 bytes of data at memory location m64 are loaded. When the register-register form of this operation is used, the lower half of the 128-bit source register is duplicated and copied into the 128-bit destination register. See Figure 3-24.



OM15997

Figure 3-24. MOVDDUP—Move One Double-FP and Duplicate

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```

IF (Source = m64)
    THEN
        (* Load instruction *)
        xmm1[63:0] = m64;
        xmm1[127:64] = m64;
    ELSE
        (* Move instruction *)
        xmm1[63:0] = xmm2[63:0];
        xmm1[127:64] = xmm2[63:0];
FI;

```

MOVDDUP (128-bit Legacy SSE version)

```

DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[VLMAX-1:128] (Unmodified)

```

VMOVDDUP (VEX.128 encoded version)

```

DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[VLMAX-1:128] ← 0

```

VMOVDDUP (VEX.256 encoded version)

```

DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[191:128] ← SRC[191:128]
DEST[255:192] ← SRC[191:128]

```

Intel C/C++ Compiler Intrinsic Equivalent

```

MOVDDUP:    __m128d _mm_movedup_pd(__m128d a)
MOVDDUP:    __m128d _mm_loaddup_pd(double const * dp)

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVDQA—Move Aligned Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Move aligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 7F /r MOVDQA <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	SSE2	Move aligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 6F /r VMOVDQA <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Move aligned packed integer values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 7F /r VMOVDQA <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move aligned packed integer values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 6F /r VMOVDQA <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move aligned packed integer values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.66.0F.WIG 7F /r VMOVDQA <i>ymm2/m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move aligned packed integer values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVDQA (128-bit load- and register- form Legacy SSE version)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

$\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

(* #GP if SRC or DEST unaligned memory operand *)

(V)MOVDQA (128-bit store forms)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

VMOVDQA (VEX.128 encoded version)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMOVDQA (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{SRC}[255:0]$

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQA: `__m128i _mm_load_si128 (__m128i *p)`

MOVDQA: `void _mm_store_si128 (__m128i *p, __m128i a)`

VMOVDQA: `__m256i _mm256_load_si256 (__m256i * p);`

VMOVDQA: `_mm256_store_si256(__m256i *p, __m256i a);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD If VEX.vvvv \neq 1111B.

MOVDQU—Move Unaligned Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Move unaligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
F3 0F 7F /r MOVDQU <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	SSE2	Move unaligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.F3.0F.WIG 6F /r VMOVDQU <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed integer values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7F /r VMOVDQU <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move unaligned packed integer values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.F3.0F.WIG 6F /r VMOVDQU <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed integer values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.F3.0F.WIG 7F /r VMOVDQU <i>ymm2/m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move unaligned packed integer values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.¹

To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVDQU load and register copy (128-bit Legacy SSE version)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

$\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

(V)MOVDQU 128-bit store-form versions

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

VMOVDQU (VEX.128 encoded version)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMOVDQU (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{SRC}[255:0]$

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQU: `void _mm_storeu_si128 (__m128i *p, __m128i a)`

MOVDQU: `__m128i _mm_loadu_si128 (__m128i *p)`

VMOVDQU: `__m256i _mm256_loadu_si256 (__m256i *p);`

VMOVDQU: `_mm256_storeu_si256(_m256i *p, __m256i a);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv \neq 1111B.

MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F2 0F D6 /r	MOVDQ2Q <i>mm, xmm</i>	RM	Valid	Valid	Move low quadword from <i>xmm</i> to <i>mmx</i> register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

DEST \leftarrow SRC[63:0];

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q: `__m64 _mm_movepi64_pi64 (__m128i a)`

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM If CR0.TS[bit 3] = 1.
 #UD If CR0.EM[bit 2] = 1.
 If CR4.OSFXSR[bit 9] = 0.
 If CPUID.01H:EDX.SSE2[bit 26] = 0.
 If the LOCK prefix is used.
 #MF If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVHPLPS— Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 12 /r MOVHPLPS <i>xmm1</i> , <i>xmm2</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from high quadword of <i>xmm2</i> to low quadword of <i>xmm1</i> .
VEX.NDS.128.OF.WIG 12 /r VMOVHPLPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of <i>xmm3</i> and low quadword of <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The high quadword of the destination operand is left unchanged. Bits (VLMAX-1:64) of the corresponding YMM destination register are unmodified.

128-bit three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (VLMAX-1:128) of the destination YMM register are zeroed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

If VMOVHPLPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVHPLPS (128-bit two-argument form)

DEST[63:0] ← SRC[127:64]

DEST[VLMAX-1:64] (Unmodified)

VMOVHPLPS (128-bit three-argument form)

DEST[63:0] ← SRC2[127:64]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPLPS: `__m128 __mm_movehl_ps(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally
#UD If VEX.L= 1.

MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD <i>xmm</i> , <i>m64</i>	RM	V/V	SSE2	Move double-precision floating-point value from <i>m64</i> to high quadword of <i>xmm</i> .
66 0F 17 /r MOVHPD <i>m64</i> , <i>xmm</i>	MR	V/V	SSE2	Move double-precision floating-point value from high quadword of <i>xmm</i> to <i>m64</i> .
VEX.128.66.0F.WIG 16 /r VMOVHPD <i>xmm2</i> , <i>xmm1</i> , <i>m64</i>	RVM	V/V	AVX	Merge double-precision floating-point value from <i>m64</i> and the low quadword of <i>xmm1</i> .
VEX.128.66.0F.WIG 17/r VMOVHPD <i>m64</i> , <i>xmm1</i>	MR	V/V	AVX	Move double-precision floating-point values from high quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64-bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) (VEX.128.66.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[VLMAX-1:128] (Unmodified)

VMOVHPD (VEX.128 encoded load)

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] ← 0

VMOVHPD (store)

DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD: __m128d _mm_loadh_pd (__m128d a, double *p)

MOVHPD: void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.

MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 16 /r MOVHPS <i>xmm, m64</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from <i>m64</i> to high quadword of <i>xmm</i> .
OF 17 /r MOVHPS <i>m64, xmm</i>	MR	V/V	SSE	Move two packed single-precision floating-point values from high quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.OF.WIG 16 /r VMOVHPS <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from <i>m64</i> and the low quadword of <i>xmm1</i> .
VEX.128.OF.WIG 17/r VMOVHPS <i>m64, xmm1</i>	MR	V/V	AVX	Move two packed single-precision floating-point values from high quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64-bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX.128 encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.OF 17 /r) is legal and has the same behavior as the existing OF 17 store.

For VMOVHPS (store) (VEX.NDS.128.OF 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)
 DEST[127:64] ← SRC[63:0]
 DEST[VLMAX-1:128] (Unmodified)

VMOVHPS (VEX.128 encoded load)

DEST[63:0] ← SRC1[63:0]
 DEST[127:64] ← SRC2[63:0]
 DEST[VLMAX-1:128] ← 0

VMOVHPS (store)

DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS: `__m128d _mm_loadh_pi (__m128d a, __m64 *p)`

MOVHPS: `void _mm_storeh_pi (__m64 *p, __m128d a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.

MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 16 /r MOVLHPS <i>xmm1</i> , <i>xmm2</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from low quadword of <i>xmm2</i> to high quadword of <i>xmm1</i> .
VEX.NDS.128.OF.WIG 16 /r VMOVLHPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of <i>xmm3</i> and low quadword of <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. The upper 128 bits of the corresponding YMM destination register are unmodified.

128-bit three-argument form

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). The upper 128-bits of the destination YMM register are zeroed.

If VMOVLHPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)
 DEST[127:64] ← SRC[63:0]
 DEST[VLMAX-1:128] (Unmodified)

VMOVLHPS (128-bit three-argument form)

DEST[63:0] ← SRC1[63:0]
 DEST[127:64] ← SRC2[63:0]
 DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS: `__m128 _mm_movelh_ps(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally
#UD If VEX.L= 1.

MOVLDP—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 12 /r MOVLDP <i>xmm</i> , <i>m64</i>	RM	V/V	SSE2	Move double-precision floating-point value from <i>m64</i> to low quadword of <i>xmm</i> register.
66 0F 13 /r MOVLDP <i>m64</i> , <i>xmm</i>	MR	V/V	SSE2	Move double-precision floating-point value from low quadword of <i>xmm</i> register to <i>m64</i> .
VEX.NDS.128.66.0F.WIG 12 /r VMOVLDP <i>xmm2</i> , <i>xmm1</i> , <i>m64</i>	RVM	V/V	AVX	Merge double-precision floating-point value from <i>m64</i> and the high quadword of <i>xmm1</i> .
VEX.128.66.0F.WIG 13/r VMOVLDP <i>m64</i> , <i>xmm1</i>	MR	V/V	AVX	Move double-precision floating-point values from low quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64-bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLDP (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLDP (store) (VEX.128.66.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVLDP is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVLDP (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] (Unmodified)

VMOVLPD (VEX.128 encoded load)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

VMOVLPD (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD: __m128d _mm_loadl_pd (__m128d a, double *p)

MOVLPD: void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.
 If VEX.vvvv ≠ 1111B.

MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 12 /r MOVLPS <i>xmm, m64</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from <i>m64</i> to low quadword of <i>xmm</i> .
0F 13 /r MOVLPS <i>m64, xmm</i>	MR	V/V	SSE	Move two packed single-precision floating-point values from low quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.0F.WIG 12 /r VMOVLPS <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from <i>m64</i> and the high quadword of <i>xmm1</i> .
VEX.128.0F.WIG 13/r VMOVLPS <i>m64, xmm1</i>	MR	V/V	AVX	Move two packed single-precision floating-point values from low quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX.128 encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (third operand), merges them with the upper 64-bits of the first source XMM register (second operand), and stores them in the low 128-bits of the destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) (VEX.128.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE load)

DEST[63:0] \leftarrow SRC[63:0]

DEST[VLMAX-1:64] (Unmodified)

VMOVLPS (VEX.128 encoded load)

DEST[63:0] \leftarrow SRC2[63:0]

DEST[127:64] \leftarrow SRC1[127:64]

DEST[VLMAX-1:128] \leftarrow 0

VMOVLPS (store)

DEST[63:0] \leftarrow SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS: `__m128 _mm_loadl_pi (__m128 a, __m64 *p)`

MOVLPS: `void _mm_storel_pi (__m64 *p, __m128 a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L= 1.
	If VEX.vvvv \neq 1111B.

MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD <i>reg, xmm</i>	RM	V/V	SSE2	Extract 2-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD <i>reg, xmm2</i>	RM	V/V	AVX	Extract 2-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD <i>reg, ymm2</i>	RM	V/V	AVX	Extract 4-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)MOVMSKPD (128-bit versions)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
IF DEST = r32
    THEN DEST[31:2] ← 0;
    ELSE DEST[63:2] ← 0;
FI
```

VMOVMSKPD (VEX.256 encoded version)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
DEST[2] ← SRC[191]
DEST[3] ← SRC[255]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD: int _mm_movemask_pd (__m128d a)

VMOVMSKPD: _mm256_movemask_pd(__m256d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv ≠ 1111B.

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 50 /r MOVMSKPS <i>reg, xmm</i>	RM	V/V	SSE	Extract 4-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.OF.WIG 50 /r VMOVMSKPS <i>reg, xmm2</i>	RM	V/V	AVX	Extract 4-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.OF.WIG 50 /r VMOVMSKPS <i>reg, ymm2</i>	RM	V/V	AVX	Extract 8-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (<i>w</i>)	ModRM:r/m (<i>r</i>)	NA	NA

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];
```

IF DEST = r32

THEN DEST[31:4] ← ZeroExtend;

ELSE DEST[63:4] ← ZeroExtend;

FI;

1. ModRM.MOD = 011B required

(V)MOVMSKPS (128-bit version)

```

DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI

```

VMOVMSKPS (VEX.256 encoded version)

```

DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_movemask_ps(__m128 a)
```

```
int _mm256_movemask_ps(__m256 a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVNTDQA — Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA <i>xmm1</i> , <i>m128</i>	RM	V/V	SSE4_1	Move double quadword from <i>m128</i> to <i>xmm</i> using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA <i>xmm1</i> , <i>m128</i>	RM	V/V	AVX	Move double quadword from <i>m128</i> to <i>xmm</i> using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA <i>ymm1</i> , <i>m256</i>	RM	V/V	AVX2	Move 256-bit data from <i>m256</i> to <i>ymm</i> using non-temporal hint if WC memory type.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(V)MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint. A processor implementation may make use of the non-temporal hint associated with this instruction if the memory source is WC (write combining) memory type. An implementation may also make use of the non-temporal hint associated with this instruction if the memory source is WB (write back) memory type.

A processor's implementation of the non-temporal hint does not override the effective memory type semantics, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Another implementation of the hint for WC memory type may optimize data transfer throughput of WC reads. A third implementation may optimize cache reads generated by (V)MOVNTDQA on WB memory type to reduce cache evictions.

WC Streaming Load Hint

For WC memory type in particular, the processor never appears to read the data into the cache hierarchy. Instead, the non-temporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a (V)MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to memory currently residing in a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in Chapter 11, "Memory Cache Control" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Because the WC protocol uses a weakly-ordered memory consistency model, an MFENCE or locked instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might reference the same WC memory locations or in order to synchronize reads of a processor with writes by other agents in the system. Because of the speculative nature of fetching due to MOVNTDQA, Streaming loads must not be used to reference memory addresses that are mapped to I/O devices having side effects or when reads to these devices are destruc-

tive. For additional information on MOVNTDQA usages, see Section 12.10.3 in Chapter 12, “Programming with SSE3, SSSE3 and SSE4” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTDQA (128bit- Legacy SSE form)

DEST \leftarrow SRC

DEST[VLMAX-1:128] (Unmodified)

VMOVNTDQA (VEX.128 encoded form)

DEST \leftarrow SRC

DEST[VLMAX-1:128] \leftarrow 0

VMOVNTDQA (VEX.256 encoded form)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVNTDQA: `__m128i _mm_stream_load_si128 (__m128i *p);`

VMOVNTDQA: `__m256i _mm256_stream_load_si256 (const __m256i *p);`

Flags Affected

None

Other Exceptions

See Exceptions Type 1.SSE4.1; additionally

#UD If VEX.L= 1.
 If VEX.vvvv \neq 1111B.

MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ <i>m128, xmm</i>	MR	V/V	SSE2	Move double quadword from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ <i>m128, xmm1</i>	MR	V/V	AVX	Move packed integer values in <i>xmm1</i> to <i>m128</i> using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ <i>m256, ymm1</i>	MR	V/V	AVX	Move packed integer values in <i>ymm1</i> to <i>m256</i> using non-temporal hint.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQ: void _mm_stream_si128(__m128i *p, __m128i a);

VMOVNTDQ: void _mm256_stream_si256(__m256i *p, __m256i a);

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

Other Exceptions

See Exceptions Type 1.SSE2; additionally
#UD If VEX.vvvv ≠ 1111B.

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C3 /r	MOVNTI <i>m32, r32</i>	MR	Valid	Valid	Move doubleword from <i>r32</i> to <i>m32</i> using non-temporal hint.
REX.W + OF C3 /r	MOVNTI <i>m64, r64</i>	MR	Valid	N.E.	Move quadword from <i>r64</i> to <i>m64</i> using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVNTI:    void _mm_stream_si32 (int *p, int a)
MOVNTI:    void _mm_stream_si64(__int64 *p, __int64 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
 #SS(0) For an illegal address in the SS segment.
 #PF(fault-code) For a page fault.
 #UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
 If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
-----------------	-------------------

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0. If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD <i>m128, xmm</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD <i>m128, xmm1</i>	MR	V/V	AVX	Move packed double-precision values in <i>xmm1</i> to <i>m128</i> using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD <i>m256, ymm1</i>	MR	V/V	AVX	Move packed double-precision values in <i>ymm1</i> to <i>m256</i> using non-temporal hint.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTPD: void _mm_stream_pd(double *p, __m128d a)

VMOVNTPD: void _mm256_stream_pd (double * p, __m256d a);

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

Other Exceptions

See Exceptions Type 1.SSE2; additionally
#UD If VEX.vvvv ≠ 1111B.

MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 2B /r MOVNTPS <i>m128, xmm</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.OF.WIG 2B /r VMOVNTPS <i>m128, xmm1</i>	MR	V/V	AVX	Move packed single-precision values <i>xmm1</i> to <i>mem</i> using non-temporal hint.
VEX.256.OF.WIG 2B /r VMOVNTPS <i>m256, ymm1</i>	MR	V/V	AVX	Move packed single-precision values <i>ymm1</i> to <i>mem</i> using non-temporal hint.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVNTPS:    void _mm_stream_ps(float * p, __m128 a)
VMOVNTPS:    void _mm256_stream_ps(float * p, __m256 a);
```

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

Other Exceptions

See Exceptions Type 1.SSE; additionally
#UD If VEX.vvvv ≠ 1111B.

MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF E7 /r	MOVNTQ <i>m64, mm</i>	MR	Valid	Valid	Move quadword from <i>mm</i> to <i>m64</i> using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ: void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 6F /r MOVQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Move quadword from <i>mm/m64</i> to <i>mm</i> .
0F 7F /r MOVQ <i>mm/m64</i> , <i>mm</i>	MR	V/V	MMX	Move quadword from <i>mm</i> to <i>mm/m64</i> .
F3 0F 7E /r MOVQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE2	Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7E /r VMOVQ <i>xmm1</i> , <i>xmm2</i>	RM	V/V	AVX	Move quadword from <i>xmm2</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7E /r VMOVQ <i>xmm1</i> , <i>m64</i>	RM	V/V	AVX	Load quadword from <i>m64</i> to <i>xmm1</i> .
66 0F D6 /r MOVQ <i>xmm2/m64</i> , <i>xmm1</i>	MR	V/V	SSE2	Move quadword from <i>xmm1</i> to <i>xmm2/mem64</i> .
VEX.128.66.0F.WIG D6 /r VMOVQ <i>xmm1/m64</i> , <i>xmm2</i>	MR	V/V	AVX	Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX.128.66.0F D6 instruction version, VEX.vvvv and VEX.L=1 are reserved and the former must be 1111b otherwise instructions will #UD.

Note: In VEX.128.F3.0F 7E version, VEX.vvvv and VEX.L=1 are reserved and the former must be 1111b, otherwise instructions will #UD.

Operation

MOVQ instruction when operating on MMX technology registers and memory locations:

DEST ← SRC;

MOVQ instruction when source and destination operands are XMM registers:

DEST[63:0] ← SRC[63:0];

DEST[127:64] ← 0000000000000000H;

MOVQ instruction when source operand is XMM register and destination operand is memory location:

$$\text{DEST} \leftarrow \text{SRC}[63:0];$$

MOVQ instruction when source operand is memory location and destination operand is XMM register:

$$\text{DEST}[63:0] \leftarrow \text{SRC};$$

$$\text{DEST}[127:64] \leftarrow 0000000000000000\text{H};$$

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[\text{VLMAX}-1:64] \leftarrow 0$$

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[\text{VLMAX}-1:64] \leftarrow 0$$

VMOVQ (7E) with memory source:

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[\text{VLMAX}-1:64] \leftarrow 0$$

VMOVQ (D6) with memory dest:

$$\text{DEST}[63:0] \leftarrow \text{SRC2}[63:0]$$

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ: `m128i _mm_mov_epi64(__m128i a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 0F D6 /r	MOVQ2DQ <i>xmm, mm</i>	RM	Valid	Valid	Move quadword from <i>mmx</i> to low quadword of <i>xmm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

DEST[63:0] ← SRC[63:0];

DEST[127:64] ← 0000000000000000H;

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ: `__128i_mm_movpi64_pi64 (__m64 a)`

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM If CR0.TS[bit 3] = 1.
 #UD If CR0.EM[bit 2] = 1.
 If CR4.OSFXSR[bit 9] = 0.
 If CPUID.01H:EDX.SSE2[bit 26] = 0.
 If the LOCK prefix is used.
 #MF If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVS/MOVS_B/MOVSW/MOVS_D/MOVSQ—Move Data from String to String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
A4	MOVS <i>m8, m8</i>	NP	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R)ESI to (R)EDI.
A5	MOVS <i>m16, m16</i>	NP	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R)ESI to (R)EDI.
A5	MOVS <i>m32, m32</i>	NP	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R)ESI to (R)EDI.
REX.W + A5	MOVS <i>m64, m64</i>	NP	Valid	N.E.	Move qword from address (R)ESI to (R)EDI.
A4	MOVS _B	NP	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R)ESI to (R)EDI.
A5	MOVSW	NP	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R)ESI to (R)EDI.
A5	MOVS _D	NP	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R)ESI to (R)EDI.
REX.W + A5	MOVSQ	NP	Valid	N.E.	Move qword from address (R)ESI to (R)EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

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

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incre-

mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with MOVS and MOVSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE/REPZ—Repeat String Operation Prefix" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST \leftarrow SRC;

Non-64-bit Mode:

```
IF (Byte move)
  THEN IF DF = 0
    THEN
      (E)SI  $\leftarrow$  (E)SI + 1;
      (E)DI  $\leftarrow$  (E)DI + 1;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 1;
      (E)DI  $\leftarrow$  (E)DI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (E)SI  $\leftarrow$  (E)SI + 2;
      (E)DI  $\leftarrow$  (E)DI + 2;
      FI;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 2;
      (E)DI  $\leftarrow$  (E)DI - 2;
    FI;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (E)SI  $\leftarrow$  (E)SI + 4;
      (E)DI  $\leftarrow$  (E)DI + 4;
      FI;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 4;
      (E)DI  $\leftarrow$  (E)DI - 4;
    FI;
  FI;
```

FI;
64-bit Mode:

```
IF (Byte move)
  THEN IF DF = 0
```

```

THEN
    (R|E)SI ← (R|E)SI + 1;
    (R|E)DI ← (R|E)DI + 1;
ELSE
    (R|E)SI ← (R|E)SI - 1;
    (R|E)DI ← (R|E)DI - 1;
FI;
ELSE IF (Word move)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 2;
        (R|E)DI ← (R|E)DI + 2;
        FI;
    ELSE
        (R|E)SI ← (R|E)SI - 2;
        (R|E)DI ← (R|E)DI - 2;
        FI;
ELSE IF (Doubleword move)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 4;
        (R|E)DI ← (R|E)DI + 4;
        FI;
    ELSE
        (R|E)SI ← (R|E)SI - 4;
        (R|E)DI ← (R|E)DI - 4;
        FI;
ELSE IF (Quadword move)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 8;
        (R|E)DI ← (R|E)DI + 8;
        FI;
    ELSE
        (R|E)SI ← (R|E)SI - 8;
        (R|E)DI ← (R|E)DI - 8;
        FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>
#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.
#UD	If the LOCK prefix is used.

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.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#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.
#UD If the LOCK prefix is used.

MOVSD—Move Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 10 /r MOVSD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE2	Move scalar double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> register.
VEX.NDS.LIG.F2.OF.WIG 10 /r VMOVSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i>	RVM	V/V	AVX	Merge scalar double-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F2.OF.WIG 10 /r VMOVSD <i>xmm1</i> , <i>m64</i>	XM	V/V	AVX	Load scalar double-precision floating-point value from <i>m64</i> to <i>xmm1</i> register.
F2 OF 11 /r MOVSD <i>xmm2/m64</i> , <i>xmm1</i>	MR	V/V	SSE2	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m64</i> .
VEX.NDS.LIG.F2.OF.WIG 11 /r VMOVSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i>	MVR	V/V	AVX	Merge scalar double-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> registers to <i>xmm1</i> .
VEX.LIG.F2.OF.WIG 11 /r VMOVSD <i>m64</i> , <i>xmm1</i>	MR	V/V	AVX	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

MOVSD moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded instruction syntax and when the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: For the “VMOVSD *m64*, *xmm1*” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b, otherwise instruction will #UD.

Note: For the “VMOVSD *xmm1*, *m64*” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low quadword of the second source operand with the quad word in bits 127:64 of the first source operand. The upper bits of the destination operand are cleared.

Operation

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] (Unmodified)

MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)

DEST[63:0] ← SRC[63:0]

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← 0

DEST[VLMAX-1:128] (Unmodified)

VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVSD: `__m128d _mm_load_sd (double *p)`

MOVSD: `void _mm_store_sd (double *p, __m128d a)`

MOVSD: `__m128d _mm_store_sd (__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVSHDUP—Move Packed Single-FP High and Duplicate

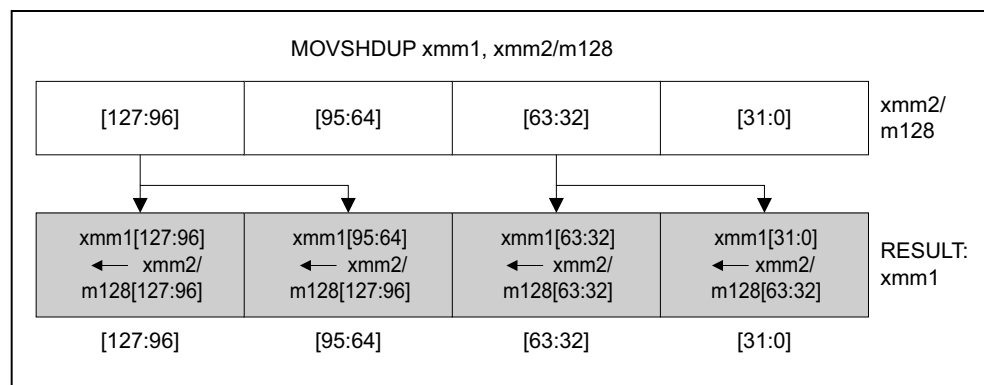
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Move two single-precision floating-point values from the higher 32-bit operand of each qword in <i>xmm2/m128</i> to <i>xmm1</i> and duplicate each 32-bit operand to the lower 32-bits of each qword.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move odd index single-precision floating-point values from <i>xmm2/mem</i> and duplicate each element into <i>xmm1</i> .
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move odd index single-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location *m128* are loaded and the single-precision elements in positions 1 and 3 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register. See Figure 3-25.



OM15998

Figure 3-25. MOVSHDUP—Move Packed Single-FP High and Duplicate

In 64-bit mode, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVSHDUP (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC}[127:96]$
 $\text{DEST}[127:96] \leftarrow \text{SRC}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VMOVSHDUP (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC}[127:96]$
 $\text{DEST}[127:96] \leftarrow \text{SRC}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMOVSHDUP (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC}[127:96]$
 $\text{DEST}[127:96] \leftarrow \text{SRC}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC}[191:160]$
 $\text{DEST}[191:160] \leftarrow \text{SRC}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC}[255:224]$
 $\text{DEST}[255:224] \leftarrow \text{SRC}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVSHDUP: `__m128 _mm_movehdup_ps(__m128 a)`
 VMOVSHDUP: `__m256 _mm256_movehdup_ps (__m256 a);`

Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions

None

Other Exceptions

See Exceptions Type 2.

MOVSLDUP—Move Packed Single-FP Low and Duplicate

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Move two single-precision floating-point values from the lower 32-bit operand of each qword in <i>xmm2/m128</i> to <i>xmm1</i> and duplicate each 32-bit operand to the higher 32-bits of each qword.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move even index single-precision floating-point values from <i>xmm2/mem</i> and duplicate each element into <i>xmm1</i> .
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move even index single-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

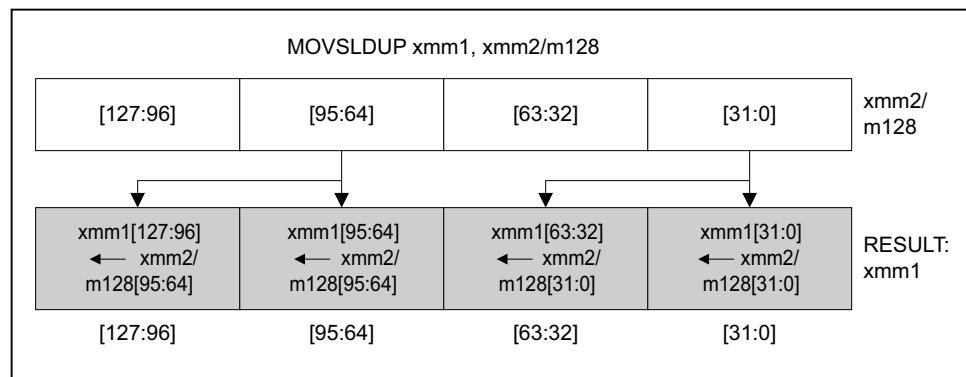
Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location *m128* are loaded and the single-precision elements in positions 0 and 2 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register.

See Figure 3-26.



OM15999

Figure 3-26. MOVSLDUP—Move Packed Single-FP Low and Duplicate

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVSLDUP (128-bit Legacy SSE version)

DEST[31:0] ← SRC[31:0]
 DEST[63:32] ← SRC[31:0]
 DEST[95:64] ← SRC[95:64]
 DEST[127:96] ← SRC[95:64]
 DEST[VLMAX-1:128] (Unmodified)

VMOVSLDUP (VEX.128 encoded version)

DEST[31:0] ← SRC[31:0]
 DEST[63:32] ← SRC[31:0]
 DEST[95:64] ← SRC[95:64]
 DEST[127:96] ← SRC[95:64]
 DEST[VLMAX-1:128] ← 0

VMOVSLDUP (VEX.256 encoded version)

DEST[31:0] ← SRC[31:0]
 DEST[63:32] ← SRC[31:0]
 DEST[95:64] ← SRC[95:64]
 DEST[127:96] ← SRC[95:64]
 DEST[159:128] ← SRC[159:128]
 DEST[191:160] ← SRC[159:128]
 DEST[223:192] ← SRC[223:192]
 DEST[255:224] ← SRC[223:192]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVSLDUP: __m128 _mm_moveldup_ps(__m128 a)
 VMOVSLDUP: __m256 _mm256_moveldup_ps (__m256 a);

Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVSS—Move Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Move scalar single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i>	RVM	V/V	AVX	Merge scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1</i> , <i>m32</i>	XM	V/V	AVX	Load scalar single-precision floating-point value from <i>m32</i> to <i>xmm1</i> register.
F3 0F 11 /r MOVSS <i>xmm2/m32</i> , <i>xmm1</i>	MR	V/V	SSE	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i>	MVR	V/V	AVX	Move scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS <i>m32</i> , <i>xmm1</i>	MR	V/V	AVX	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded syntax and when the source and destination operands are XMM registers, the high doublewords of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high doublewords of the destination operand is cleared to all 0s.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low dword of the second source operand with three dwords in bits 127:32 of the first source operand. The upper bits of the destination operand are cleared.

Note: For the “VMOVSS *m32*, *xmm1*” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the “VMOVSS *xmm1*, *m32*” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)

$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$

$\text{DEST}[\text{VLMAX}-1:32]$ (Unmodified)

MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)

$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$

MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)

$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$

$\text{DEST}[127:32] \leftarrow 0$

$\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)

$\text{DEST}[31:0] \leftarrow \text{SRC2}[31:0]$

$\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)

$\text{DEST}[31:0] \leftarrow \text{SRC2}[31:0]$

$\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)

$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$

$\text{DEST}[\text{VLMAX}-1:32] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MOVSS: `__m128 _mm_load_ss(float * p)`

MOVSS: `void _mm_store_ss(float * p, __m128 a)`

MOVSS: `__m128 _mm_move_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv \neq 1111B.

MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF BE /r	MOVSX r16, r/m8	RM	Valid	Valid	Move byte to word with sign-extension.
OF BE /r	MOVSX r32, r/m8	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX + OF BE /r	MOVSX r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.
OF BF /r	MOVSX r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + OF BF /r	MOVSX r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
REX.W** + 63 /r	MOVSXD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** The use of MOVSXD without REX.W in 64-bit mode is discouraged, Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← SignExtend(SRC);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 10 /r VMOVUPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed double-precision floating-point from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.256.66.0F.WIG 10 /r VMOVUPD <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed double-precision floating-point from <i>ymm2/mem</i> to <i>ymm1</i> .
66 0F 11 /r MOVUPD <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 11 /r VMOVUPD <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move unaligned packed double-precision floating-point from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 11 /r VMOVUPD <i>ymm2/m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move unaligned packed double-precision floating-point from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions:

Moves a double quadword containing two packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.¹

To move double-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPD instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVUPD (128-bit load and register-copy form Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

(V)MOVUPD (128-bit store form)

DEST[127:0] ← SRC[127:0]

VMOVUPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] ← 0

VMOVUPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVUPD: `__m128 _mm_loadu_pd(double * p)`

MOVUPD: `void _mm_storeu_pd(double *p, __m128 a)`

VMOVUPD: `__m256d _mm256_loadu_pd (__m256d * p);`

VMOVUPD: `_mm256_storeu_pd(_m256d *p, __m256d a);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 10 /r MOVUPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.128.OF.WIG 10 /r VMOVUPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed single-precision floating-point from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.256.OF.WIG 10 /r VMOVUPS <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed single-precision floating-point from <i>ymm2/mem</i> to <i>ymm1</i> .
OF 11 /r MOVUPS <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.OF.WIG 11 /r VMOVUPS <i>xmm2/m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move unaligned packed single-precision floating-point from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.OF.WIG 11 /r VMOVUPS <i>ymm2/m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move unaligned packed single-precision floating-point from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions: Moves a double quadword containing four packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.¹

To move packed single-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPS instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

Operation

MOVUPS (128-bit load and register-copy form Legacy SSE version)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

$\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

(V)MOVUPS (128-bit store form)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

VMOVUPS (VEX.128 encoded load-form)

$\text{DEST}[127:0] \leftarrow \text{SRC}[127:0]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMOVUPS (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{SRC}[255:0]$

Intel C/C++ Compiler Intrinsic Equivalent

MOVUPS: `__m128 _mm_loadu_ps(double * p)`

MOVUPS: `void _mm_storeu_ps(double *p, __m128 a)`

VMOVUPS: `__m256 _mm256_loadu_ps (__m256 * p);`

VMOVUPS: `_mm256_storeu_ps(_m256 *p, __m256 a);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD If VEX.vvvv \neq 1111B.

MOVZX—Move with Zero-Extend

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF B6 /r	MOVZX <i>r16, r/m8</i>	RM	Valid	Valid	Move byte to word with zero-extension.
OF B6 /r	MOVZX <i>r32, r/m8</i>	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + OF B6 /r	MOVZX <i>r64, r/m8*</i>	RM	Valid	N.E.	Move byte to quadword, zero-extension.
OF B7 /r	MOVZX <i>r32, r/m16</i>	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + OF B7 /r	MOVZX <i>r64, r/m16</i>	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← ZeroExtend(SRC);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

MPSADBW — Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm1</i> and <i>xmm2/m128</i> are determined by <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 42 /r ib VMPSADBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .
VEX.NDS.256.66.0F3A.WIG 42 /r ib VMPSADBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>ymm2</i> and <i>ymm3/m128</i> and writes the results in <i>ymm1</i> . Starting offsets within <i>ymm2</i> and <i>ymm3/m128</i> are determined by <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L = 1, an attempt to execute the instruction encoded with VEX.L = 1 will cause an #UD exception.

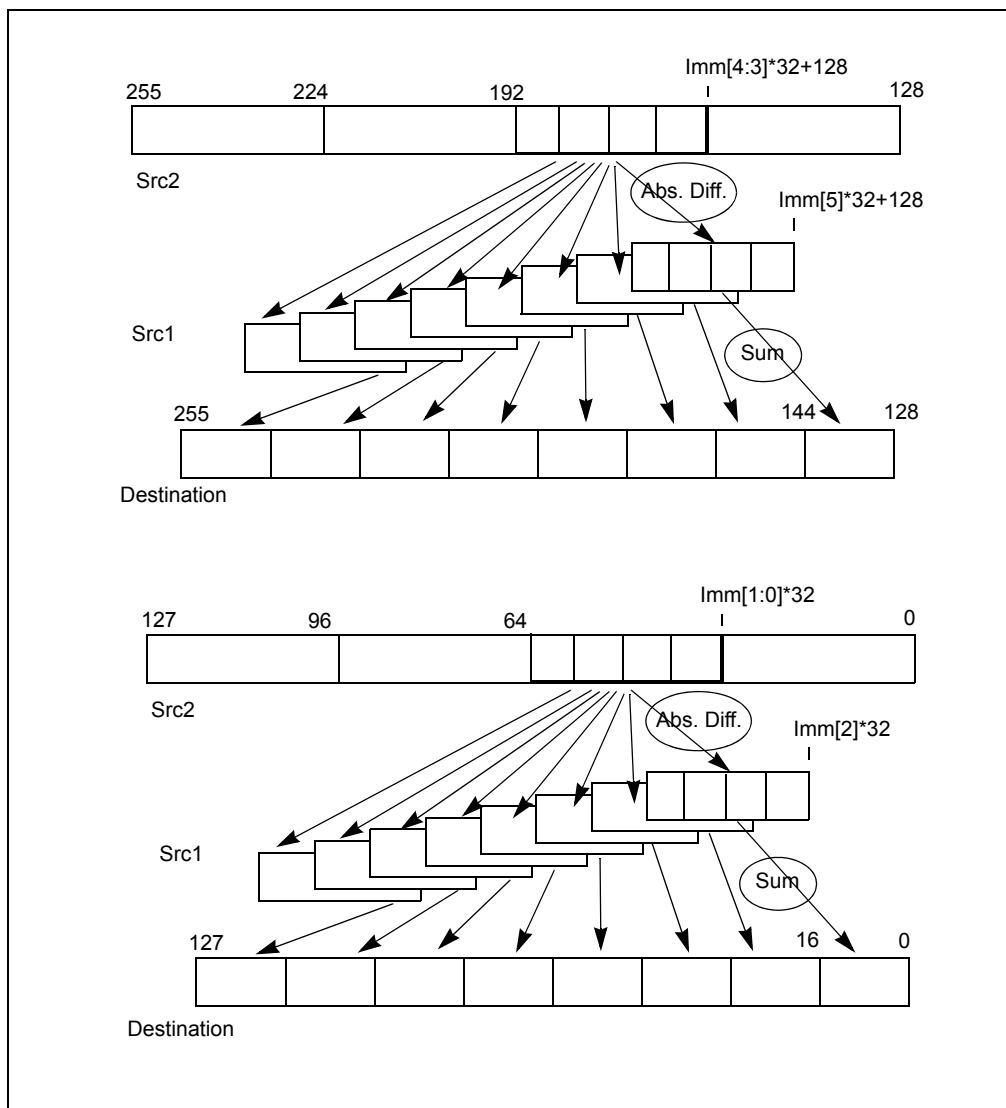


Figure 3-27. 256-bit VMPSADBW Operation

Operation

VMPSADBW (VEX.256 encoded version)

$\text{BLK2_OFFSET} \leftarrow \text{imm8}[1:0] * 32$

$\text{BLK1_OFFSET} \leftarrow \text{imm8}[2] * 32$

$\text{SRC1_BYTE0} \leftarrow \text{SRC1}[\text{BLK1_OFFSET}+7:\text{BLK1_OFFSET}]$

$\text{SRC1_BYTE1} \leftarrow \text{SRC1}[\text{BLK1_OFFSET}+15:\text{BLK1_OFFSET}+8]$

$\text{SRC1_BYTE2} \leftarrow \text{SRC1}[\text{BLK1_OFFSET}+23:\text{BLK1_OFFSET}+16]$

$\text{SRC1_BYTE3} \leftarrow \text{SRC1}[\text{BLK1_OFFSET}+31:\text{BLK1_OFFSET}+24]$

$\text{SRC1_BYTE4} \leftarrow \text{SRC1}[\text{BLK1_OFFSET}+39:\text{BLK1_OFFSET}+32]$

$\text{SRC1_BYTE5} \leftarrow \text{SRC1}[\text{BLK1_OFFSET}+47:\text{BLK1_OFFSET}+40]$

$SRC1_BYTE6 \leftarrow SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]$
 $SRC1_BYTE7 \leftarrow SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]$
 $SRC1_BYTE8 \leftarrow SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]$
 $SRC1_BYTE9 \leftarrow SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]$
 $SRC1_BYTE10 \leftarrow SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]$
 $SRC2_BYTE0 \leftarrow SRC2[BLK2_OFFSET+7:BLK2_OFFSET]$
 $SRC2_BYTE1 \leftarrow SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]$
 $SRC2_BYTE2 \leftarrow SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]$
 $SRC2_BYTE3 \leftarrow SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]$

$TEMP0 \leftarrow ABS(SRC1_BYTE0 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE3)$
 $DEST[15:0] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE3)$
 $DEST[31:16] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE3)$
 $DEST[47:32] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE3)$
 $DEST[63:48] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE3)$
 $DEST[79:64] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE3)$
 $DEST[95:80] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE3)$
 $DEST[111:96] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE1)$

$TEMP2 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE10 - SRC2_BYTE3)$
 $DEST[127:112] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$BLK2_OFFSET \leftarrow imm8[4:3] * 32 + 128$
 $BLK1_OFFSET \leftarrow imm8[5] * 32 + 128$
 $SRC1_BYTE0 \leftarrow SRC1[BLK1_OFFSET+7:BLK1_OFFSET]$
 $SRC1_BYTE1 \leftarrow SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]$
 $SRC1_BYTE2 \leftarrow SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]$
 $SRC1_BYTE3 \leftarrow SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]$
 $SRC1_BYTE4 \leftarrow SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]$
 $SRC1_BYTE5 \leftarrow SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]$
 $SRC1_BYTE6 \leftarrow SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]$
 $SRC1_BYTE7 \leftarrow SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]$
 $SRC1_BYTE8 \leftarrow SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]$
 $SRC1_BYTE9 \leftarrow SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]$
 $SRC1_BYTE10 \leftarrow SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]$

$SRC2_BYTE0 \leftarrow SRC2[BLK2_OFFSET+7:BLK2_OFFSET]$
 $SRC2_BYTE1 \leftarrow SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]$
 $SRC2_BYTE2 \leftarrow SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]$
 $SRC2_BYTE3 \leftarrow SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]$

$TEMP0 \leftarrow ABS(SRC1_BYTE0 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE3)$
 $DEST[143:128] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE3)$
 $DEST[159:144] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE3)$
 $DEST[175:160] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE3)$
 $DEST[191:176] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE3)$
 $DEST[207:192] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE0)$

$TEMP1 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE3)$
 $DEST[223:208] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE3)$
 $DEST[239:224] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE10 - SRC2_BYTE3)$
 $DEST[255:240] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

VMPSADBW (VEX.128 encoded version)

$BLK2_OFFSET \leftarrow imm8[1:0] * 32$
 $BLK1_OFFSET \leftarrow imm8[2] * 32$
 $SRC1_BYTE0 \leftarrow SRC1[BLK1_OFFSET+7:BLK1_OFFSET]$
 $SRC1_BYTE1 \leftarrow SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]$
 $SRC1_BYTE2 \leftarrow SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]$
 $SRC1_BYTE3 \leftarrow SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]$
 $SRC1_BYTE4 \leftarrow SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]$
 $SRC1_BYTE5 \leftarrow SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]$
 $SRC1_BYTE6 \leftarrow SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]$
 $SRC1_BYTE7 \leftarrow SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]$
 $SRC1_BYTE8 \leftarrow SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]$
 $SRC1_BYTE9 \leftarrow SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]$
 $SRC1_BYTE10 \leftarrow SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]$

$SRC2_BYTE0 \leftarrow SRC2[BLK2_OFFSET+7:BLK2_OFFSET]$
 $SRC2_BYTE1 \leftarrow SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]$
 $SRC2_BYTE2 \leftarrow SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]$
 $SRC2_BYTE3 \leftarrow SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]$

$TEMP0 \leftarrow ABS(SRC1_BYTE0 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE3)$
 $DEST[15:0] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE3)$
 $DEST[31:16] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE3)$
 $DEST[47:32] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE3)$
 $DEST[63:48] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE3)$
 $DEST[79:64] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE3)$
 $DEST[95:80] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE3)$
 $DEST[111:96] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE10 - SRC2_BYTE3)$
 $DEST[127:112] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $DEST[VLMAX-1:128] \leftarrow 0$

MPSADBW (128-bit Legacy SSE version)

$SRC_OFFSET \leftarrow imm8[1:0] * 32$
 $DEST_OFFSET \leftarrow imm8[2] * 32$
 $DEST_BYTE0 \leftarrow DEST[DEST_OFFSET+7:DEST_OFFSET]$
 $DEST_BYTE1 \leftarrow DEST[DEST_OFFSET+15:DEST_OFFSET+8]$
 $DEST_BYTE2 \leftarrow DEST[DEST_OFFSET+23:DEST_OFFSET+16]$
 $DEST_BYTE3 \leftarrow DEST[DEST_OFFSET+31:DEST_OFFSET+24]$
 $DEST_BYTE4 \leftarrow DEST[DEST_OFFSET+39:DEST_OFFSET+32]$
 $DEST_BYTE5 \leftarrow DEST[DEST_OFFSET+47:DEST_OFFSET+40]$
 $DEST_BYTE6 \leftarrow DEST[DEST_OFFSET+55:DEST_OFFSET+48]$
 $DEST_BYTE7 \leftarrow DEST[DEST_OFFSET+63:DEST_OFFSET+56]$
 $DEST_BYTE8 \leftarrow DEST[DEST_OFFSET+71:DEST_OFFSET+64]$
 $DEST_BYTE9 \leftarrow DEST[DEST_OFFSET+79:DEST_OFFSET+72]$
 $DEST_BYTE10 \leftarrow DEST[DEST_OFFSET+87:DEST_OFFSET+80]$

$SRC_BYTE0 \leftarrow SRC[SRC_OFFSET+7:SRC_OFFSET]$
 $SRC_BYTE1 \leftarrow SRC[SRC_OFFSET+15:SRC_OFFSET+8]$
 $SRC_BYTE2 \leftarrow SRC[SRC_OFFSET+23:SRC_OFFSET+16]$
 $SRC_BYTE3 \leftarrow SRC[SRC_OFFSET+31:SRC_OFFSET+24]$

$TEMP0 \leftarrow ABS(DEST_BYTE0 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE1 - SRC_BYTE1)$

TEMP2 \leftarrow ABS(DEST_BYTE2 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE3)
 DEST[15:0] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE1 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE2 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE3)
 DEST[31:16] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE2 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE3)
 DEST[47:32] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE3 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE3)
 DEST[63:48] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE4 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE3)
 DEST[79:64] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE5 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE8 - SRC_BYTE3)
 DEST[95:80] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE6 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE8 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE9 - SRC_BYTE3)
 DEST[111:96] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO \leftarrow ABS(DEST_BYTE7 - SRC_BYTE0)
 TEMP1 \leftarrow ABS(DEST_BYTE8 - SRC_BYTE1)
 TEMP2 \leftarrow ABS(DEST_BYTE9 - SRC_BYTE2)
 TEMP3 \leftarrow ABS(DEST_BYTE10 - SRC_BYTE3)
 DEST[127:112] \leftarrow TEMPO + TEMP1 + TEMP2 + TEMP3
 DEST[VLMAX-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)MPSADBW: `__m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);`

VMPSADBW: `__m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);`

Flags Affected

None

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

MUL—Unsigned Multiply

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /4	MUL <i>r/m8</i>	M	Valid	Valid	Unsigned multiply ($AX \leftarrow AL * r/m8$).
REX + F6 /4	MUL <i>r/m8</i> *	M	Valid	N.E.	Unsigned multiply ($AX \leftarrow AL * r/m8$).
F7 /4	MUL <i>r/m16</i>	M	Valid	Valid	Unsigned multiply ($DX:AX \leftarrow AX * r/m16$).
F7 /4	MUL <i>r/m32</i>	M	Valid	Valid	Unsigned multiply ($EDX:EAX \leftarrow EAX * r/m32$).
REX.W + F7 /4	MUL <i>r/m64</i>	M	Valid	N.E.	Unsigned multiply ($RDX:RAX \leftarrow RAX * r/m64$).

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 3-65.

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

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Table 3-65. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	<i>r/m8</i>	AX
Word	AX	<i>r/m16</i>	DX:AX
Doubleword	EAX	<i>r/m32</i>	EDX:EAX
Quadword	RAX	<i>r/m64</i>	RDX:RAX

Operation

```

IF (Byte operation)
    THEN
         $AX \leftarrow AL * SRC;$ 
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                 $DX:AX \leftarrow AX * SRC;$ 
            ELSE IF OperandSize = 32
                THEN  $EDX:EAX \leftarrow EAX * SRC;$  FI;
            ELSE (* OperandSize = 64 *)
                 $RDX:RAX \leftarrow RAX * SRC;$ 
        FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.

MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 59 /r MULPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply packed double-precision floating-point values in <i>xmm2/m128</i> by <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 59 /r VMULPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 59 /r VMULPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Multiply packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply of the two or four packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MULPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[127:64] ← DEST[127:64] * SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VMULPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VMULPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[191:128] ← SRC1[191:128] * SRC2[191:128]
DEST[255:192] ← SRC1[255:192] * SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

MULPD: `__m128d _mm_mul_pd (m128d a, m128d b)`

VMULPD: `__m256d _mm256_mul_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2

MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 59 /r MULPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Multiply packed single-precision floating-point values in <i>xmm2/mem</i> by <i>xmm1</i> .
VEX.NDS.128.OF.WIG 59 /r VMULPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply packed single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.OF.WIG 59 /r VMULPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MULPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VMULPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```


VMULPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] * \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] * \text{SRC2}[255:224]$.

Intel C/C++ Compiler Intrinsic Equivalent

MULPS: `__m128 _mm_mul_ps(__m128 a, __m128 b)`
 VMULPS: `__m256 _mm256_mul_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2

MULSD—Multiply Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 59 /r MULSD <i>xmm1</i> , <i>xmm2/mem64</i>	RM	V/V	SSE2	Multiply the low double-precision floating-point value in <i>xmm2/mem64</i> by low double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 59/r VMULSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/mem64</i>	RVM	V/V	AVX	Multiply the low double-precision floating-point value in <i>xmm3/mem64</i> by low double-precision floating-point value in <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low double-precision floating-point value in the source operand (second operand) by the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MULSD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] * SRC[63:0]

DEST[VLMAX-1:64] (Unmodified)

VMULSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MULSD: `__m128d _mm_mul_sd (m128d a, m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3

MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 59 /r MULSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Multiply the low single-precision floating-point value in <i>xmm2/mem</i> by the low single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Multiply the low single-precision floating-point value in <i>xmm3/mem</i> by the low single-precision floating-point value in <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low single-precision floating-point value from the source operand (second operand) by the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MULSS (128-bit Legacy SSE version)

DEST[31:0] ← DEST[31:0] * SRC[31:0]

DEST[VLMAX-1:32] (Unmodified)

VMULSS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MULSS: `__m128 __mm_mul_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3

MULX — Unsigned Multiply Without Affecting Flags

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.F2.0F38.W0 F6 /r MULX <i>r32a, r32b, r/m32</i>	RVM	V/V	BMI2	Unsigned multiply of <i>r/m32</i> with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F38.W1 F6 /r MULX <i>r64a, r64b, r/m64</i>	RVM	V/N.E.	BMI2	Unsigned multiply of <i>r/m64</i> with RDX without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
    DEST2 ← (SRC1*SRC2)[63:0];
    DEST1 ← (SRC1*SRC2)[127:64];
FI
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

```
unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);
```

```
unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

MWAIT—Monitor Wait

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 C9	MWAIT	NP	Valid	Valid	A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized “waiting” states and will select among those states based on the hint argument.

Table 3-66 describes the meaning of ECX and EAX registers for MWAIT extensions.

Table 3-66. MWAIT Extension Register (ECX)

Bits	Description
0	Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF=0). May be set only if CPUID.05H:ECX[bit 1] = 1.
31: 1	Reserved

Table 3-67. MWAIT Hints Register (EAX)

Bits	Description
3 : 0	Sub C-state within a C-state, indicated by bits [7:4]
7 : 4	Target C-state* Value of 0 means C1; 1 means C2 and so on Value of 01111B means C0 Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states
31: 8	Reserved

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 14, “Power and Thermal Management,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Operation

```
(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension
MWAIT EAX, ECX *)
{
  WHILE ( ("Monitor Hardware is in armed state")) {
    implementation_dependent_optimized_state(EAX, ECX);
  }
  Set the state of Monitor Hardware as triggered;
}
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MWAIT:      void _mm_mwait(unsigned extensions, unsigned hints)
```

Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF ( !trigger_store_happened ) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0) If ECX[31:1] ≠ 0.
If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.

#UD If CPUID.01H:ECX.MONITOR[bit 3] = 0.
If current privilege level is not 0.

Real Address Mode Exceptions

#GP If ECX[31:1] ≠ 0.
If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.

#UD If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If RCX[63:1] ≠ 0.
If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.

#UD If the current privilege level is not 0.
If CPUID.01H:ECX.MONITOR[bit 3] = 0.

NEG—Two's Complement Negation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /3	NEG <i>r/m8</i>	M	Valid	Valid	Two's complement negate <i>r/m8</i> .
REX + F6 /3	NEG <i>r/m8</i> *	M	Valid	N.E.	Two's complement negate <i>r/m8</i> .
F7 /3	NEG <i>r/m16</i>	M	Valid	Valid	Two's complement negate <i>r/m16</i> .
F7 /3	NEG <i>r/m32</i>	M	Valid	Valid	Two's complement negate <i>r/m32</i> .
REX.W + F7 /3	NEG <i>r/m64</i>	M	Valid	N.E.	Two's complement negate <i>r/m64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (<i>r</i> , <i>w</i>)	NA	NA	NA

Description

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF DEST = 0
    THEN CF ← 0;
    ELSE CF ← 1;
FI;
DEST ← [- (DEST)]
```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

NOP—No Operation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
90	NOP	NP	Valid	Valid	One byte no-operation instruction.
0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

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

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of “no operation” as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-9. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

NOT—One's Complement Negation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /2	NOT <i>r/m8</i>	M	Valid	Valid	Reverse each bit of <i>r/m8</i> .
REX + F6 /2	NOT <i>r/m8</i> *	M	Valid	N.E.	Reverse each bit of <i>r/m8</i> .
F7 /2	NOT <i>r/m16</i>	M	Valid	Valid	Reverse each bit of <i>r/m16</i> .
F7 /2	NOT <i>r/m32</i>	M	Valid	Valid	Reverse each bit of <i>r/m32</i> .
REX.W + F7 /2	NOT <i>r/m64</i>	M	Valid	N.E.	Reverse each bit of <i>r/m64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	NA	NA	NA

Description

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← NOT DEST;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

OR—Logical Inclusive OR

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	I	Valid	Valid	AL OR <i>imm8</i> .
0D <i>iw</i>	OR AX, <i>imm16</i>	I	Valid	Valid	AX OR <i>imm16</i> .
0D <i>id</i>	OR EAX, <i>imm32</i>	I	Valid	Valid	EAX OR <i>imm32</i> .
REX.W + 0D <i>id</i>	OR RAX, <i>imm32</i>	I	Valid	N.E.	RAX OR <i>imm32</i> (sign-extended).
80 /1 <i>ib</i>	OR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> OR <i>imm8</i> .
REX + 80 /1 <i>ib</i>	OR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> OR <i>imm8</i> .
81 /1 <i>iw</i>	OR <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> OR <i>imm16</i> .
81 /1 <i>id</i>	OR <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> OR <i>imm32</i> .
REX.W + 81 /1 <i>id</i>	OR <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> OR <i>imm32</i> (sign-extended).
83 /1 <i>ib</i>	OR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> OR <i>imm8</i> (sign-extended).
83 /1 <i>ib</i>	OR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> OR <i>imm8</i> (sign-extended).
REX.W + 83 /1 <i>ib</i>	OR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> OR <i>imm8</i> (sign-extended).
08 /r	OR <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> OR <i>r8</i> .
REX + 08 /r	OR <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m8</i> OR <i>r8</i> .
09 /r	OR <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> OR <i>r16</i> .
09 /r	OR <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> OR <i>r32</i> .
REX.W + 09 /r	OR <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> OR <i>r64</i> .
0A /r	OR <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> OR <i>r/m8</i> .
REX + 0A /r	OR <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r8</i> OR <i>r/m8</i> .
0B /r	OR <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> OR <i>r/m16</i> .
0B /r	OR <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> OR <i>r/m32</i> .
REX.W + 0B /r	OR <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> OR <i>r/m64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/16/32</i>	NA	NA
MI	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	<i>imm8/16/32</i>	NA	NA
MR	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	ModRM: <i>reg</i> (<i>r</i>)	NA	NA
RM	ModRM: <i>reg</i> (<i>r</i> , <i>w</i>)	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA

Description

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← DEST OR SRC;

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 56 /r ORPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 56 /r VORPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.OF.WIG 56 /r VORPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: If VORPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

ORPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
 DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]
 DEST[VLMAX-1:128] (Unmodified)

VORPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
 DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
 DEST[VLMAX-1:128] ← 0

VORPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
 DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
 DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
 DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]

Intel® C/C++ Compiler Intrinsic Equivalent

ORPD: __m128d _mm_or_pd(__m128d a, __m128d b);

VORPD: __m256d _mm256_or_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 56 /r ORPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Bitwise OR of <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.OF.WIG 56 /r VORPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 56 /r VORPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 Encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: If VORPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

ORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VORPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE OR } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE OR } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE OR } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE OR } \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE OR } \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE OR } \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE OR } \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE OR } \text{SRC2}[255:224]$.

Intel C/C++ Compiler Intrinsic Equivalent

ORPS: `__m128 _mm_or_ps (__m128 a, __m128 b);`
 VORPS: `__m256 _mm256_or_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

OUT—Output to Port

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
E6 <i>ib</i>	OUT <i>imm8</i> , AL	I	Valid	Valid	Output byte in AL to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , AX	I	Valid	Valid	Output word in AX to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , EAX	I	Valid	Valid	Output doubleword in EAX to I/O port address <i>imm8</i> .
EE	OUT DX, AL	NP	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	NP	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	NP	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	<i>imm8</i>	NA	NA	NA
NP	NA	NA	NA	NA

Description

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

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

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

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 16, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

Operation

```

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Writes to selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Writes to selected I/O port *)
  FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

Same as protected mode exceptions.

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

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
6E	OUTS DX, <i>m8</i>	NP	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m16</i>	NP	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m32</i>	NP	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6E	OUTSB	NP	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSW	NP	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSD	NP	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.

NOTES:

* See IA-32 Architecture Compatibility section below.

** In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

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

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

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

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ /REPNE/REP NZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 16, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is support using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Writes to I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Writes to I/O port *)
    FI;
```

Byte transfer:

```
IF 64-bit mode
    Then
        IF 64-Bit Address Size
            THEN
                IF DF = 0
                    THEN RSI ← RSI + 1;
                    ELSE RSI ← RSI - 1;
                FI;
            ELSE (* 32-Bit Address Size *)
                IF DF = 0
                    THEN ESI ← ESI + 1;
                    ELSE ESI ← ESI - 1;
                FI;
            FI;
        ELSE
            IF DF = 0
                THEN (ESI) ← (ESI) + 1;
                ELSE (ESI) ← (ESI) - 1;
            FI;
        FI;
```

Word transfer:

```
IF 64-bit mode
```

```

Then
  IF 64-Bit Address Size
    THEN
      IF DF = 0
        THEN RSI ← RSI + 2;
        ELSE RSI ← RSI or - 2;
      FI;
    ELSE (* 32-Bit Address Size *)
      IF DF = 0
        THEN ESI ← ESI + 2;
        ELSE ESI ← ESI - 2;
      FI;
    FI;
  ELSE
    IF DF = 0
      THEN (ESI) ← (ESI) + 2;
      ELSE (ESI) ← (ESI) - 2;
    FI;
  FI;
Doubleword transfer:
  IF 64-bit mode
    Then
      IF 64-Bit Address Size
        THEN
          IF DF = 0
            THEN RSI ← RSI + 4;
            ELSE RSI ← RSI or - 4;
          FI;
        ELSE (* 32-Bit Address Size *)
          IF DF = 0
            THEN ESI ← ESI + 4;
            ELSE ESI ← ESI - 4;
          FI;
        FI;
      ELSE
        IF DF = 0
          THEN (ESI) ← (ESI) + 4;
          ELSE (ESI) ← (ESI) - 4;
        FI;
      FI;
    FI;

```

Flags Affected

None.

Protected Mode Exceptions

- | | |
|-----------------|--|
| #GP(0) | <p>If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.</p> <p>If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment.</p> <p>If the segment register contains a NULL segment selector.</p> |
| #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. |

#UD If the LOCK prefix is used.

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.
 #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
 #PF(fault-code) If a page fault occurs.
 #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
 #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
 #GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
 If the memory address is in a non-canonical form.
 #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.
 #UD If the LOCK prefix is used.

PABSBB/PABSW/PABSD — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 1C /r ¹ PABSBB mm1, mm2/m64	RM	V/V	SSSE3	Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1C /r PABSBB xmm1, xmm2/m128	RM	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
0F 38 1D /r ¹ PABSW mm1, mm2/m64	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
0F 38 1E /r ¹ PABSD mm1, mm2/m64	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSBB xmm1, xmm2/m128	RM	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	RM	V/V	AVX	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	RM	V/V	AVX	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.256.66.0F38.WIG 1C /r VPABSBB ymm1, ymm2/m256	RM	V/V	AVX2	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1D /r VPABSW ymm1, ymm2/m256	RM	V/V	AVX2	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1E /r VPABSD ymm1, ymm2/m256	RM	V/V	AVX2	Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(V)PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). (V)PABSB operates on signed bytes, (V)PABSW operates on 16-bit words, and (V)PABSD operates on signed 32-bit integers. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register, a YMM register, a 128-bit memory location, or a 256-bit memory location. The destination operand can be an MMX, an XMM or a YMM register. Both operands can be MMX registers or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The source operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

PABSB (with 64 bit operands)

Unsigned DEST[7:0] ← ABS(SRC[7:0])
 Repeat operation for 2nd through 7th bytes
 Unsigned DEST[63:56] ← ABS(SRC[63:56])

PABSB (with 128 bit operands)

Unsigned DEST[7:0] ← ABS(SRC[7:0])
 Repeat operation for 2nd through 15th bytes
 Unsigned DEST[127:120] ← ABS(SRC[127:120])

PABSW (with 64 bit operands)

Unsigned DEST[15:0] ← ABS(SRC[15:0])
 Repeat operation for 2nd through 3rd 16-bit words
 Unsigned DEST[63:48] ← ABS(SRC[63:48])

PABSW (with 128 bit operands)

Unsigned DEST[15:0] ← ABS(SRC[15:0])
 Repeat operation for 2nd through 7th 16-bit words
 Unsigned DEST[127:112] ← ABS(SRC[127:112])

PABSD (with 64 bit operands)

Unsigned DEST[31:0] ← ABS(SRC[31:0])
 Unsigned DEST[63:32] ← ABS(SRC[63:32])

PABSD (with 128 bit operands)

Unsigned DEST[31:0] ← ABS(SRC[31:0])
 Repeat operation for 2nd through 3rd 32-bit double words
 Unsigned DEST[127:96] ← ABS(SRC[127:96])

PABSB (128-bit Legacy SSE version)

DEST[127:0] ← BYTE_ABS(SRC)
 DEST[VLMAX-1:128] (Unmodified)

VPASB (VEX.128 encoded version)

DEST[127:0] ← BYTE_ABS(SRC)

DEST[VLMAX-1:128] ← 0

VPASB (VEX.256 encoded version)

Unsigned DEST[7:0] ← ABS(SRC[7:0])

Repeat operation for 2nd through 31st bytes

Unsigned DEST[255:248] ← ABS(SRC[255:248])

PABSW (128-bit Legacy SSE version)

DEST[127:0] ← WORD_ABS(SRC)

DEST[VLMAX-1:128] (Unmodified)

VPABSW (VEX.128 encoded version)

DEST[127:0] ← WORD_ABS(SRC)

DEST[VLMAX-1:128] ← 0

VPABSW (VEX.256 encoded version)

Unsigned DEST[15:0] ← ABS(SRC[15:0])

Repeat operation for 2nd through 15th 16-bit words

Unsigned DEST[255:240] ← ABS(SRC[255:240])

PABSD (128-bit Legacy SSE version)

DEST[127:0] ← DWORD_ABS(SRC)

DEST[VLMAX-1:128] (Unmodified)

VPABSD (VEX.128 encoded version)

DEST[127:0] ← DWORD_ABS(SRC)

DEST[VLMAX-1:128] ← 0

VPABSD (VEX.256 encoded version)

Unsigned DEST[31:0] ← ABS(SRC[31:0])

Repeat operation for 2nd through 7th 32-bit double words

Unsigned DEST[255:224] ← ABS(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalents

PABSB: __m64 _mm_abs_pi8 (__m64 a)

(V)PABSB: __m128i _mm_abs_epi8 (__m128i a)

VPABSB: __m256i _mm256_abs_epi8 (__m256i a)

PABSW: __m64 _mm_abs_pi16 (__m64 a)

(V)PABSW: __m128i _mm_abs_epi16 (__m128i a)

VPABSW: __m256i _mm256_abs_epi16 (__m256i a)

PABSD: __m64 _mm_abs_pi32 (__m64 a)

(V)PABSD: __m128i _mm_abs_epi32 (__m128i a)

VPABSD: __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 63 /r ¹ PACKSSWB <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	MMX	Converts 4 packed signed word integers from <i>mm1</i> and from <i>mm2/m64</i> into 8 packed signed byte integers in <i>mm1</i> using signed saturation.
66 OF 63 /r PACKSSWB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
OF 6B /r ¹ PACKSSDW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	MMX	Converts 2 packed signed doubleword integers from <i>mm1</i> and from <i>mm2/m64</i> into 4 packed signed word integers in <i>mm1</i> using signed saturation.
66 OF 6B /r PACKSSDW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.OF.WIG 63 /r VPACKSSWB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Converts 8 packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.OF.WIG 6B /r VPACKSSDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Converts 4 packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.256.66.OF.WIG 63 /r VPACKSSWB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Converts 16 packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 32 packed signed byte integers in <i>ymm1</i> using signed saturation.
VEX.NDS.256.66.OF.WIG 6B /r VPACKSSDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Converts 8 packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 16 packed signed word integers in <i>ymm1</i> using signed saturation.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-2 for an example of the packing operation.

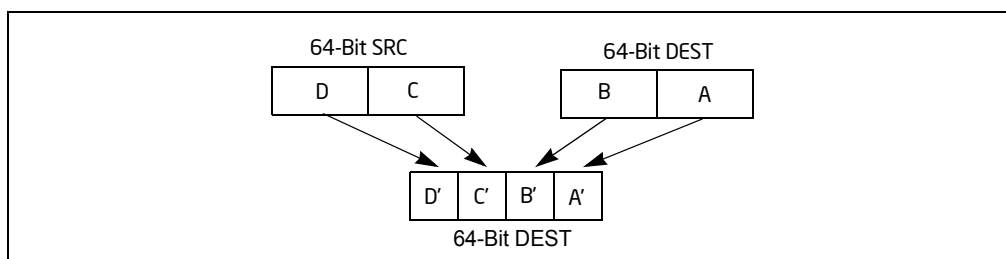


Figure 4-2. Operation of the PACKSSDW Instruction Using 64-bit Operands

The (V)PACKSSWB instruction converts 4, 8 or 16 signed word integers from the destination operand (first operand) and 4, 8 or 16 signed word integers from the source operand (second operand) into 8, 16 or 32 signed byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

The (V)PACKSSDW instruction packs 2, 4 or 8 signed doublewords from the destination operand (first operand) and 2, 4 or 8 signed doublewords from the source operand (second operand) into 4, 8 or 16 signed words in the destination operand (see Figure 4-2). If a signed doubleword integer value is beyond the range of a signed word (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed word integer value of 7FFFH or 8000H, respectively, is stored into the destination.

The (V)PACKSSWB and (V)PACKSSDW instructions operate on either 64-bit, 128-bit operands or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PACKSSWB (with 64-bit operands)

```

DEST[7:0] ← SaturateSignedWordToSignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToSignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToSignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToSignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToSignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToSignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToSignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToSignedByte SRC[63:48];

```

PACKSSDW (with 64-bit operands)

```

DEST[15:0] ← SaturateSignedDoublewordToSignedWord DEST[31:0];
DEST[31:16] ← SaturateSignedDoublewordToSignedWord DEST[63:32];
DEST[47:32] ← SaturateSignedDoublewordToSignedWord SRC[31:0];
DEST[63:48] ← SaturateSignedDoublewordToSignedWord SRC[63:32];

```

PACKSSWB instruction (128-bit Legacy SSE version)

DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
 DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
 DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
 DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
 DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
 DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
 DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
 DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);

PACKSSDW instruction (128-bit Legacy SSE version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
 DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
 DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
 DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);

VPACKSSWB instruction (VEX.128 encoded version)

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
 DEST[VLMAX-1:128] ← 0;

VPACKSSDW instruction (VEX.128 encoded version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);

DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
 DEST[VLMAX-1:128] ← 0;

VPACKSSWB instruction (VEX.256 encoded version)

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
 DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);
 DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
 DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
 DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
 DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
 DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
 DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
 DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
 DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
 DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
 DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
 DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
 DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
 DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
 DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
 DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);

VPACKSSDW instruction (VEX.256 encoded version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
 DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
 DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
 DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
 DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
 DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);
 DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);
 DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);

DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);

Intel C/C++ Compiler Intrinsic Equivalents

PACKSSWB: __m64 _mm_packs_pi16(__m64 m1, __m64 m2)
 (V)PACKSSWB: __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
 VPACKSSWB: __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
 PACKSSDW: __m64 _mm_packs_pi32 (__m64 m1, __m64 m2)
 (V)PACKSSDW: __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
 VPACKSSDW: __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PACKUSDW — Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Convert 4 packed signed doubleword integers from <i>xmm1</i> and 4 packed signed doubleword integers from <i>xmm2/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F38.WIG 2B /r VPACKUSDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Convert 4 packed signed doubleword integers from <i>xmm2</i> and 4 packed signed doubleword integers from <i>xmm3/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F38.WIG 2B /r VPACKUSDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Convert 8 packed signed doubleword integers from <i>ymm2</i> and 8 packed signed doubleword integers from <i>ymm3/m128</i> into 16 packed unsigned word integers in <i>ymm1</i> using unsigned saturation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts packed signed doubleword integers into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PACKUSDW (Legacy SSE instruction)

```

TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] ← (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] ← (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];

```

$\text{DEST}[63:48] \leftarrow (\text{SRC}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[79:64];$
 $\text{TMP}[95:80] \leftarrow (\text{SRC}[63:32] < 0) ? 0 : \text{SRC}[47:32];$
 $\text{DEST}[95:80] \leftarrow (\text{SRC}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[95:80];$
 $\text{TMP}[111:96] \leftarrow (\text{SRC}[95:64] < 0) ? 0 : \text{SRC}[79:64];$
 $\text{DEST}[111:96] \leftarrow (\text{SRC}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[111:96];$
 $\text{TMP}[127:112] \leftarrow (\text{SRC}[127:96] < 0) ? 0 : \text{SRC}[111:96];$
 $\text{DEST}[127:112] \leftarrow (\text{SRC}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[127:112];$

PACKUSDW (VEX.128 encoded version)

$\text{TMP}[15:0] \leftarrow (\text{SRC1}[31:0] < 0) ? 0 : \text{SRC1}[15:0];$
 $\text{DEST}[15:0] \leftarrow (\text{SRC1}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[15:0];$
 $\text{TMP}[31:16] \leftarrow (\text{SRC1}[63:32] < 0) ? 0 : \text{SRC1}[47:32];$
 $\text{DEST}[31:16] \leftarrow (\text{SRC1}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[31:16];$
 $\text{TMP}[47:32] \leftarrow (\text{SRC1}[95:64] < 0) ? 0 : \text{SRC1}[79:64];$
 $\text{DEST}[47:32] \leftarrow (\text{SRC1}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[47:32];$
 $\text{TMP}[63:48] \leftarrow (\text{SRC1}[127:96] < 0) ? 0 : \text{SRC1}[111:96];$
 $\text{DEST}[63:48] \leftarrow (\text{SRC1}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[63:48];$
 $\text{TMP}[79:64] \leftarrow (\text{SRC2}[31:0] < 0) ? 0 : \text{SRC2}[15:0];$
 $\text{DEST}[63:48] \leftarrow (\text{SRC2}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[79:64];$
 $\text{TMP}[95:80] \leftarrow (\text{SRC2}[63:32] < 0) ? 0 : \text{SRC2}[47:32];$
 $\text{DEST}[95:80] \leftarrow (\text{SRC2}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[95:80];$
 $\text{TMP}[111:96] \leftarrow (\text{SRC2}[95:64] < 0) ? 0 : \text{SRC2}[79:64];$
 $\text{DEST}[111:96] \leftarrow (\text{SRC2}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[111:96];$
 $\text{TMP}[127:112] \leftarrow (\text{SRC2}[127:96] < 0) ? 0 : \text{SRC2}[111:96];$
 $\text{DEST}[127:112] \leftarrow (\text{SRC2}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[127:112];$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0;$

VPACKUSDW (VEX.256 encoded version)

$\text{TMP}[15:0] \leftarrow (\text{SRC1}[31:0] < 0) ? 0 : \text{SRC1}[15:0];$
 $\text{DEST}[15:0] \leftarrow (\text{SRC1}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[15:0];$
 $\text{TMP}[31:16] \leftarrow (\text{SRC1}[63:32] < 0) ? 0 : \text{SRC1}[47:32];$
 $\text{DEST}[31:16] \leftarrow (\text{SRC1}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[31:16];$
 $\text{TMP}[47:32] \leftarrow (\text{SRC1}[95:64] < 0) ? 0 : \text{SRC1}[79:64];$
 $\text{DEST}[47:32] \leftarrow (\text{SRC1}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[47:32];$
 $\text{TMP}[63:48] \leftarrow (\text{SRC1}[127:96] < 0) ? 0 : \text{SRC1}[111:96];$
 $\text{DEST}[63:48] \leftarrow (\text{SRC1}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[63:48];$
 $\text{TMP}[79:64] \leftarrow (\text{SRC2}[31:0] < 0) ? 0 : \text{SRC2}[15:0];$
 $\text{DEST}[63:48] \leftarrow (\text{SRC2}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[79:64];$
 $\text{TMP}[95:80] \leftarrow (\text{SRC2}[63:32] < 0) ? 0 : \text{SRC2}[47:32];$
 $\text{DEST}[95:80] \leftarrow (\text{SRC2}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[95:80];$
 $\text{TMP}[111:96] \leftarrow (\text{SRC2}[95:64] < 0) ? 0 : \text{SRC2}[79:64];$
 $\text{DEST}[111:96] \leftarrow (\text{SRC2}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[111:96];$
 $\text{TMP}[127:112] \leftarrow (\text{SRC2}[127:96] < 0) ? 0 : \text{SRC2}[111:96];$
 $\text{DEST}[128:112] \leftarrow (\text{SRC2}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[127:112];$
 $\text{TMP}[143:128] \leftarrow (\text{SRC1}[159:128] < 0) ? 0 : \text{SRC1}[143:128];$
 $\text{DEST}[143:128] \leftarrow (\text{SRC1}[159:128] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[143:128];$
 $\text{TMP}[159:144] \leftarrow (\text{SRC1}[191:160] < 0) ? 0 : \text{SRC1}[175:160];$
 $\text{DEST}[159:144] \leftarrow (\text{SRC1}[191:160] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[159:144];$
 $\text{TMP}[175:160] \leftarrow (\text{SRC1}[223:192] < 0) ? 0 : \text{SRC1}[207:192];$
 $\text{DEST}[175:160] \leftarrow (\text{SRC1}[223:192] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[175:160];$
 $\text{TMP}[191:176] \leftarrow (\text{SRC1}[255:224] < 0) ? 0 : \text{SRC1}[239:224];$
 $\text{DEST}[191:176] \leftarrow (\text{SRC1}[255:224] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[191:176];$
 $\text{TMP}[207:192] \leftarrow (\text{SRC2}[159:128] < 0) ? 0 : \text{SRC2}[143:128];$
 $\text{DEST}[207:192] \leftarrow (\text{SRC2}[159:128] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[207:192];$

```

TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];

```

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PACKUSDW: __m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW:   __m256i _mm256_packus_epi32(__m256i m1, __m256i m2);

```

Flags Affected

None.

SIMD Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 67 /r ¹ PACKUSWB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Converts 4 signed word integers from <i>mm</i> and 4 signed word integers from <i>mm/m64</i> into 8 unsigned byte integers in <i>mm</i> using unsigned saturation.
66 0F 67 /r PACKUSWB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Converts 8 signed word integers from <i>xmm1</i> and 8 signed word integers from <i>xmm2/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Converts 8 signed word integers from <i>xmm2</i> and 8 signed word integers from <i>xmm3/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Converts 16 signed word integers from <i>ymm2</i> and 16 signed word integers from <i>ymm3/m256</i> into 32 unsigned byte integers in <i>ymm1</i> using unsigned saturation.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts 4, 8 or 16 signed word integers from the destination operand (first operand) and 4, 8 or 16 signed word integers from the source operand (second operand) into 8, 16 or 32 unsigned byte integers and stores the result in the destination operand. (See Figure 4-2 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

The PACKUSWB instruction operates on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

PACKUSWB (with 64-bit operands)

```

DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];

```

PACKUSWB (Legacy SSE instruction)

```

DEST[7:0] ← SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);

```

PACKUSWB (VEX.128 encoded version)

```

DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[VLMAX-1:128] ← 0;

```

VPACKUSWB (VEX.256 encoded version)

```

DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);

```

DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
 DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
 DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
 DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
 DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
 DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
 DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
 DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
 DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
 DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
 DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
 DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
 DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
 DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
 DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
 DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
 DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
 DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
 DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
 DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
 DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
 DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
 DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
 DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
 DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
 DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
 DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);

Intel C/C++ Compiler Intrinsic Equivalent

PACKUSWB: `__m64 _mm_packs_pu16(__m64 m1, __m64 m2)`
 (V)PACKUSWB: `__m128i _mm_packus_epi16(__m128i m1, __m128i m2)`
 VPACKUSWB: `__m256i _mm256_packus_epi16(__m256i m1, __m256i m2);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDB/PADDW/PADD—Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FC /r ¹ PADDB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed byte integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FC /r PADDB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> .
0F FD /r ¹ PADDW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed word integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FD /r PADDW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed word integers from <i>xmm2/m128</i> and <i>xmm1</i> .
0F FE /r ¹ PADDD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed doubleword integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FE /r PADDD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed doubleword integers from <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG FC /r VPADDB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed byte integers from <i>xmm3/m128</i> and <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG FD /r VPADDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed word integers from <i>xmm3/m128</i> and <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG FE /r VPADDD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed doubleword integers from <i>xmm3/m128</i> and <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG FC /r VPADDB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG FD /r VPADDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed word integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG FE /r VPADDD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed doubleword integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

Adds the packed byte, word, doubleword, or quadword integers in the first source operand to the second source operand and stores the result in the destination operand. When a result is too large to be represented in the

8/16/32 integer (overflow), the result is wrapped around and the low bits are written to the destination element (that is, the carry is ignored).

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

PADDB (with 128-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 14th byte *)
DEST[127:120] ← DEST[111:120] + SRC[127:120];
```

VPADDB (VEX.128 encoded version)

```
DEST[7:0] ← SRC1[7:0] + SRC2[7:0]
DEST[15:8] ← SRC1[15:8] + SRC2[15:8]
DEST[23:16] ← SRC1[23:16] + SRC2[23:16]
DEST[31:24] ← SRC1[31:24] + SRC2[31:24]
DEST[39:32] ← SRC1[39:32] + SRC2[39:32]
DEST[47:40] ← SRC1[47:40] + SRC2[47:40]
DEST[55:48] ← SRC1[55:48] + SRC2[55:48]
DEST[63:56] ← SRC1[63:56] + SRC2[63:56]
DEST[71:64] ← SRC1[71:64] + SRC2[71:64]
DEST[79:72] ← SRC1[79:72] + SRC2[79:72]
DEST[87:80] ← SRC1[87:80] + SRC2[87:80]
DEST[95:88] ← SRC1[95:88] + SRC2[95:88]
DEST[103:96] ← SRC1[103:96] + SRC2[103:96]
DEST[111:104] ← SRC1[111:104] + SRC2[111:104]
DEST[119:112] ← SRC1[119:112] + SRC2[119:112]
DEST[127:120] ← SRC1[127:120] + SRC2[127:120]
DEST[VLMAX-1:128] ← 0
```

VPADDB (VEX.256 encoded instruction)

```
DEST[7:0] ← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248] ← SRC1[255:248] + SRC2[255:248];
```

PADDW (with 64-bit operands)

DEST[15:0] \leftarrow DEST[15:0] + SRC[15:0];
 (* Repeat add operation for 2nd and 3th word *)
 DEST[63:48] \leftarrow DEST[63:48] + SRC[63:48];

PADDW (with 128-bit operands)

DEST[15:0] \leftarrow DEST[15:0] + SRC[15:0];
 (* Repeat add operation for 2nd through 7th word *)
 DEST[127:112] \leftarrow DEST[127:112] + SRC[127:112];

VPADDW (VEX.128 encoded version)

DEST[15:0] \leftarrow SRC1[15:0] + SRC2[15:0]
 DEST[31:16] \leftarrow SRC1[31:16] + SRC2[31:16]
 DEST[47:32] \leftarrow SRC1[47:32] + SRC2[47:32]
 DEST[63:48] \leftarrow SRC1[63:48] + SRC2[63:48]
 DEST[79:64] \leftarrow SRC1[79:64] + SRC2[79:64]
 DEST[95:80] \leftarrow SRC1[95:80] + SRC2[95:80]
 DEST[111:96] \leftarrow SRC1[111:96] + SRC2[111:96]
 DEST[127:112] \leftarrow SRC1[127:112] + SRC2[127:112]
 DEST[VLMAX-1:128] \leftarrow 0

VPADDW (VEX.256 encoded instruction)

DEST[15:0] \leftarrow SRC1[15:0] + SRC2[15:0];
 (* Repeat add operation for 2nd through 15th word *)
 DEST[255:240] \leftarrow SRC1[255:240] + SRC2[255:240];

PADDD (with 64-bit operands)

DEST[31:0] \leftarrow DEST[31:0] + SRC[31:0];
 DEST[63:32] \leftarrow DEST[63:32] + SRC[63:32];

PADDD (with 128-bit operands)

DEST[31:0] \leftarrow DEST[31:0] + SRC[31:0];
 (* Repeat add operation for 2nd and 3th doubleword *)
 DEST[127:96] \leftarrow DEST[127:96] + SRC[127:96];

VPADDD (VEX.128 encoded version)

DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0]
 DEST[63:32] \leftarrow SRC1[63:32] + SRC2[63:32]
 DEST[95:64] \leftarrow SRC1[95:64] + SRC2[95:64]
 DEST[127:96] \leftarrow SRC1[127:96] + SRC2[127:96]
 DEST[VLMAX-1:128] \leftarrow 0

VPADDD (VEX.256 encoded instruction)

DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0];
 (* Repeat add operation for 2nd and 7th doubleword *)
 DEST[255:224] \leftarrow SRC1[255:224] + SRC2[255:224];

Intel C/C++ Compiler Intrinsic Equivalents

PADDB: __m64 _mm_add_pi8(__m64 m1, __m64 m2)
 (V)PADDB: __m128i _mm_add_epi8 (__m128ia, __m128ib)
 VPADDB: __m256i _mm256_add_epi8 (__m256ia, __m256i b)
 PADDW: __m64 _mm_add_pi16(__m64 m1, __m64 m2)
 (V)PADDW: __m128i _mm_add_epi16 (__m128i a, __m128i b)

VPADDW: __m256i _mm256_add_epi16 (__m256i a, __m256i b)
 PADD: __m64 _mm_add_pi32(__m64 m1, __m64 m2)
 (V)PADD: __m128i _mm_add_epi32 (__m128i a, __m128i b)
 VPADD: __m256i _mm256_add_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDQ—Add Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D4 /r ¹ PADDQ mm1, mm2/m64	RM	V/V	SSE2	Add quadword integer mm2/m64 to mm1.
66 0F D4 /r PADDQ xmm1, xmm2/m128	RM	V/V	SSE2	Add packed quadword integers xmm2/m128 to xmm1.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed quadword integers xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed quadword integers from ymm2, ymm3/m256 and store in ymm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the first operand (destination operand) to the second operand (source operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, a SIMD add is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PADDQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDQ (with 64-Bit operands)

DEST[63:0] ← DEST[63:0] + SRC[63:0];

PADDQ (with 128-Bit operands)

DEST[63:0] \leftarrow DEST[63:0] + SRC[63:0];
 DEST[127:64] \leftarrow DEST[127:64] + SRC[127:64];

VPADDQ (VEX.128 encoded instruction)

DEST[63:0] \leftarrow SRC1[63:0] + SRC2[63:0];
 DEST[127:64] \leftarrow SRC1[127:64] + SRC2[127:64];
 DEST[VLMAX-1:128] \leftarrow 0;

VPADDQ (VEX.256 encoded instruction)

DEST[63:0] \leftarrow SRC1[63:0] + SRC2[63:0];
 DEST[127:64] \leftarrow SRC1[127:64] + SRC2[127:64];
 DEST[191:128] \leftarrow SRC1[191:128] + SRC2[191:128];
 DEST[255:192] \leftarrow SRC1[255:192] + SRC2[255:192];

Intel C/C++ Compiler Intrinsic Equivalents

PADDQ: __m64 _mm_add_si64 (__m64 a, __m64 b)
 (V)PADDQ: __m128i _mm_add_epi64 (__m128i a, __m128i b)
 VPADDQ: __m256i _mm256_add_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EC /r ¹ PADDSB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed signed byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F EC /r PADDSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed signed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.
0F ED /r ¹ PADDSW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed signed word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F ED /r PADDSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed signed word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed signed byte integers from <i>xmm3/m128</i> and <i>xmm2</i> and saturate the results.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed signed word integers from <i>xmm3/m128</i> and <i>xmm2</i> and saturate the results.
VEX.NDS.256.66.0F.WIG EC /r VPADDSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG ED /r VPADDSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The PADDSB instruction adds packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PADDSW instruction adds packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56]);
```

PADDSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

VPADDSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48]);
```

PADDSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

VPADDSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```


Intel C/C++ Compiler Intrinsic Equivalents

```
PADDSSB:    __m64 _mm_adds_pi8(__m64 m1, __m64 m2)
(V)PADDSSB: __m128i _mm_adds_epi8 ( __m128i a, __m128i b)
VPADDSSB:   __m256i _mm256_adds_epi8 ( __m256i a, __m256i b)
PADDSSW:    __m64 _mm_adds_pi16(__m64 m1, __m64 m2)
(V)PADDSSW: __m128i _mm_adds_epi16 ( __m128i a, __m128i b)
VPADDSSW:   __m256i _mm256_adds_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF DC /r ¹ PADDUSB mm, mm/m64	RM	V/V	MMX	Add packed unsigned byte integers from mm/m64 and mm and saturate the results.
66 OF DC /r PADDUSB xmm1, xmm2/m128	RM	V/V	SSE2	Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.
OF DD /r ¹ PADDUSW mm, mm/m64	RM	V/V	MMX	Add packed unsigned word integers from mm/m64 and mm and saturate the results.
66 OF DD /r PADDUSW xmm1, xmm2/m128	RM	V/V	SSE2	Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.
VEX.NDS.128.66.0F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

The (V)PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

The (V)PADDUSW instruction adds packed unsigned word integers. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX tech-

nology register or a 64-bit memory location. In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56])
```

PADDUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

VPADDUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] ← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48]);
```

PADDUSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

VPADDUSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB: __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
 PADDUSW: __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
 (V)PADDUSB: __m128i _mm_adds_epu8 (__m128i a, __m128i b)
 (V)PADDUSW: __m128i _mm_adds_epu16 (__m128i a, __m128i b)
 VPADDUSB: __m256i _mm256_adds_epu8 (__m256i a, __m256i b)
 VPADDUSW: __m256i _mm256_adds_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 3A 0F /r ib ¹ PALIGNR <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>mm1</i> .
66 OF 3A 0F /r ib PALIGNR <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG OF /r ib VPALIGNR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Concatenate <i>xmm2</i> and <i>xmm3/m128</i> , extract byte aligned result shifted to the right by constant value in <i>imm8</i> and result is stored in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG OF /r ib VPALIGNR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX2	Concatenate pairs of 16 bytes in <i>ymm2</i> and <i>ymm3/m256</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and two 16-byte results are stored in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX, XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The *imm8*[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source

operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

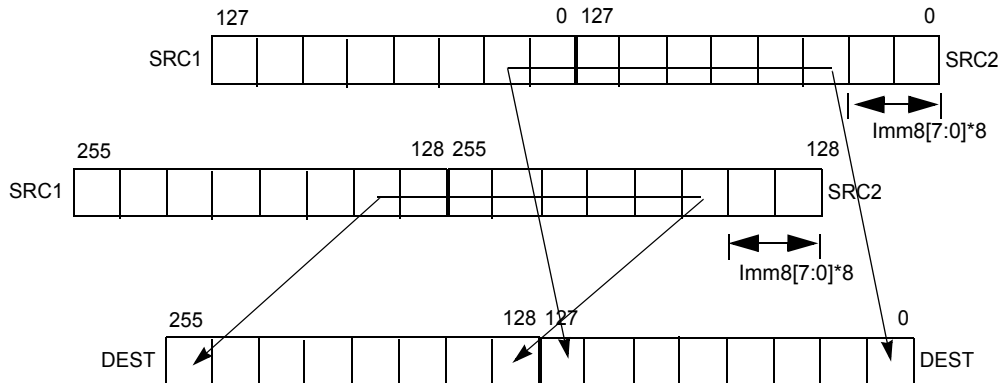


Figure 4-3. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST, SRC) >> (imm8*8)
DEST[63:0] = temp1[63:0]
```

PALIGNR (with 128-bit operands)

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0]) >> (imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] (Unmodified)
```

VPALIGNR (VEX.128 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0]) >> (imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] ← 0
```

VPALIGNR (VEX.256 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0]) >> (imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128]) >> (imm8[7:0]*8);
DEST[255:128] ← temp1[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalents

```
PALIGNR:      __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
(V)PALIGNR:   __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
VPALIGNR:     __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DB /r ¹ PAND <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Bitwise AND <i>mm/m64</i> and <i>mm</i> .
66 0F DB /r PAND <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise AND of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG DB /r VPAND <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Bitwise AND of <i>xmm3/m128</i> and <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG DB /r VPAND <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Bitwise AND of <i>ymm2</i> , and <i>ymm3/m256</i> and store result in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PAND (128-bit Legacy SSE version)

DEST ← DEST AND SRC

DEST[VLMAX-1:128] (Unmodified)

VPAND (VEX.128 encoded version)

DEST ← SRC1 AND SRC2

DEST[VLMAX-1:128] ← 0

VPAND (VEX.256 encoded instruction)
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])

Intel C/C++ Compiler Intrinsic Equivalent

PAND: __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND: __m128i _mm_and_si128 (__m128i a, __m128i b)
VPAND: __m256i _mm256_and_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DF /r ¹ PANDN <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Bitwise AND NOT of <i>mm/m64</i> and <i>mm</i> .
66 0F DF /r PANDN <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG DF /r VPANDN <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Bitwise AND NOT of <i>xmm3/m128</i> and <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG DF /r VPANDN <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Bitwise AND NOT of <i>ymm2</i> , and <i>ymm3/m256</i> and store result in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PANDN(128-bit Legacy SSE version)

DEST ← NOT(DEST) AND SRC

DEST[VLMAX-1:128] (Unmodified)

VPANDN (VEX.128 encoded version)

DEST ← NOT(SRC1) AND SRC2

DEST[VLMAX-1:128] ← 0

VPANDN (VEX.256 encoded instruction) $\text{DEST}[255:0] \leftarrow ((\text{NOT SRC1}[255:0]) \text{ AND SRC2}[255:0])$ **Intel C/C++ Compiler Intrinsic Equivalent**PANDN: `__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)`(V)PANDN: `__m128i _mm_andnot_si128 (__m128i a, __m128i b)`VPANDN: `__m256i _mm256_andnot_si256 (__m256i a, __m256i b)`**Flags Affected**

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PAUSE—Spin Loop Hint

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 90	PAUSE	NP	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

Execute_Next_Instruction(Delay);

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E0 /r ¹ PAVGB <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSE	Average packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E0, /r PAVGB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Average packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
0F E3 /r ¹ PAVGW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSE	Average packed unsigned word integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E3 /r PAVGW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Average packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
VEX.NDS.128.66.0F.WIG E0 /r VPAVGB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Average packed unsigned byte integers from <i>xmm3/m128</i> and <i>xmm2</i> with rounding.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Average packed unsigned word integers from <i>xmm3/m128</i> and <i>xmm2</i> with rounding.
VEX.NDS.256.66.0F.WIG E0 /r VPAVGB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG E3 /r VPAVGW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

PAVGB (with 64-bit operands)

$DEST[7:0] \leftarrow (SRC[7:0] + DEST[7:0] + 1) \gg 1$; (* Temp sum before shifting is 9 bits *)
 (* Repeat operation performed for bytes 2 through 6 *)
 $DEST[63:56] \leftarrow (SRC[63:56] + DEST[63:56] + 1) \gg 1$;

PAVGW (with 64-bit operands)

$DEST[15:0] \leftarrow (SRC[15:0] + DEST[15:0] + 1) \gg 1$; (* Temp sum before shifting is 17 bits *)
 (* Repeat operation performed for words 2 and 3 *)
 $DEST[63:48] \leftarrow (SRC[63:48] + DEST[63:48] + 1) \gg 1$;

PAVGB (with 128-bit operands)

$DEST[7:0] \leftarrow (SRC[7:0] + DEST[7:0] + 1) \gg 1$; (* Temp sum before shifting is 9 bits *)
 (* Repeat operation performed for bytes 2 through 14 *)
 $DEST[127:120] \leftarrow (SRC[127:120] + DEST[127:120] + 1) \gg 1$;

PAVGW (with 128-bit operands)

$DEST[15:0] \leftarrow (SRC[15:0] + DEST[15:0] + 1) \gg 1$; (* Temp sum before shifting is 17 bits *)
 (* Repeat operation performed for words 2 through 6 *)
 $DEST[127:112] \leftarrow (SRC[127:112] + DEST[127:112] + 1) \gg 1$;

VPAVGB (VEX.128 encoded version)

$DEST[7:0] \leftarrow (SRC1[7:0] + SRC2[7:0] + 1) \gg 1$;
 (* Repeat operation performed for bytes 2 through 15 *)
 $DEST[127:120] \leftarrow (SRC1[127:120] + SRC2[127:120] + 1) \gg 1$;
 $DEST[VLMAX-1:128] \leftarrow 0$

VPAVGW (VEX.128 encoded version)

$DEST[15:0] \leftarrow (SRC1[15:0] + SRC2[15:0] + 1) \gg 1$;
 (* Repeat operation performed for 16-bit words 2 through 7 *)
 $DEST[127:112] \leftarrow (SRC1[127:112] + SRC2[127:112] + 1) \gg 1$;
 $DEST[VLMAX-1:128] \leftarrow 0$

VPAVGB (VEX.256 encoded instruction)

$DEST[7:0] \leftarrow (SRC1[7:0] + SRC2[7:0] + 1) \gg 1$; (* Temp sum before shifting is 9 bits *)
 (* Repeat operation performed for bytes 2 through 31)
 $DEST[255:248] \leftarrow (SRC1[255:248] + SRC2[255:248] + 1) \gg 1$;

VPAVGW (VEX.256 encoded instruction)

$DEST[15:0] \leftarrow (SRC1[15:0] + SRC2[15:0] + 1) \gg 1$; (* Temp sum before shifting is 17 bits *)
 (* Repeat operation performed for words 2 through 15)
 $DEST[255:14] \leftarrow (SRC1[255:240] + SRC2[255:240] + 1) \gg 1$;

Intel C/C++ Compiler Intrinsic Equivalent

PAVGB: `__m64 _mm_avg_pu8 (__m64 a, __m64 b)`
 PAVGW: `__m64 _mm_avg_pu16 (__m64 a, __m64 b)`
 (V)PAVGB: `__m128i _mm_avg_epu8 (__m128i a, __m128i b)`

(V)PAVGW: __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB: __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW: __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PBLENDVB — Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RM	V/V	SSE4_1	Select byte values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in the high bit of each byte in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4C /r /is4 VPBLENDVB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Select byte values from <i>xmm2</i> and <i>xmm3/m128</i> using mask bits in the specified mask register, <i>xmm4</i> , and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.W0 4C /r /is4 VPBLENDVB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX2	Select byte values from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in the high bit of each byte in <i>ymm4</i> and store the values into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	< <i>XMM0</i> >	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, *XMM0*. The mask bits are the most significant bit in each byte element of the *XMM0* register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register *XMM0*.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register *XMM0*. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (VLMAX-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats *XMM0* implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

PBLENDVB (128-bit Legacy SSE version)

MASK ← *XMM0*

IF (MASK[7] = 1) THEN DEST[7:0] ← SRC[7:0];

ELSE DEST[7:0] ← DEST[7:0];

IF (MASK[15] = 1) THEN DEST[15:8] ← SRC[15:8];


```

ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120])
DEST[VLMAX-1:128] (Unmodified)

```

VPBLENDVB (VEX.128 encoded version)

```

MASK ← SRC3
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC2[87:80]

```

```

ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
DEST[VLMAX-1:128] ← 0

```

VPBLENDVB (VEX.256 encoded version)

```

MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
IF (MASK[135] == 1) THEN DEST[135:128] ← SRC2[135:128];
ELSE DEST[135:128] ← SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144]
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152]

```

```

ELSE DEST[159:152] ← SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] ← SRC2[167:160]
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] ← SRC2[183:176]
ELSE DEST[183:176] ← SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] ← SRC2[191:184]
ELSE DEST[191:184] ← SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] ← SRC2[199:192]
ELSE DEST[199:192] ← SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] ← SRC2[207:200]
ELSE DEST[207:200] ← SRC1[207:200];
IF (MASK[215] == 1) THEN DEST[215:208] ← SRC2[215:208]
ELSE DEST[215:208] ← SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] ← SRC2[223:216]
ELSE DEST[223:216] ← SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] ← SRC2[231:224]
ELSE DEST[231:224] ← SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] ← SRC2[239:232]
ELSE DEST[239:232] ← SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] ← SRC2[247:240]
ELSE DEST[247:240] ← SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] ← SRC2[255:248]
ELSE DEST[255:248] ← SRC1[255:248]

```

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PBLENDVB:  __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
VPBLENDVB:    __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

```

#UD                If VEX.L = 1.
                   If VEX.W = 1.

```

PBLENDW — Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select words from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0E /r ib VPBLENDW <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select words from <i>xmm2</i> and <i>xmm3/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0E /r ib VPBLENDW <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Select words from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in <i>imm8</i> and store the values into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

PBLENDW (128-bit Legacy SSE version)

```

IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC[127:112]

```

ELSE DEST[127:112] \leftarrow DEST[127:112]

VPBLENDW (VEX.128 encoded version)

IF (imm8[0] = 1) THEN DEST[15:0] \leftarrow SRC2[15:0]
 ELSE DEST[15:0] \leftarrow SRC1[15:0]
 IF (imm8[1] = 1) THEN DEST[31:16] \leftarrow SRC2[31:16]
 ELSE DEST[31:16] \leftarrow SRC1[31:16]
 IF (imm8[2] = 1) THEN DEST[47:32] \leftarrow SRC2[47:32]
 ELSE DEST[47:32] \leftarrow SRC1[47:32]
 IF (imm8[3] = 1) THEN DEST[63:48] \leftarrow SRC2[63:48]
 ELSE DEST[63:48] \leftarrow SRC1[63:48]
 IF (imm8[4] = 1) THEN DEST[79:64] \leftarrow SRC2[79:64]
 ELSE DEST[79:64] \leftarrow SRC1[79:64]
 IF (imm8[5] = 1) THEN DEST[95:80] \leftarrow SRC2[95:80]
 ELSE DEST[95:80] \leftarrow SRC1[95:80]
 IF (imm8[6] = 1) THEN DEST[111:96] \leftarrow SRC2[111:96]
 ELSE DEST[111:96] \leftarrow SRC1[111:96]
 IF (imm8[7] = 1) THEN DEST[127:112] \leftarrow SRC2[127:112]
 ELSE DEST[127:112] \leftarrow SRC1[127:112]
 DEST[VLMAX-1:128] \leftarrow 0

VPBLENDW (VEX.256 encoded version)

IF (imm8[0] == 1) THEN DEST[15:0] \leftarrow SRC2[15:0]
 ELSE DEST[15:0] \leftarrow SRC1[15:0]
 IF (imm8[1] == 1) THEN DEST[31:16] \leftarrow SRC2[31:16]
 ELSE DEST[31:16] \leftarrow SRC1[31:16]
 IF (imm8[2] == 1) THEN DEST[47:32] \leftarrow SRC2[47:32]
 ELSE DEST[47:32] \leftarrow SRC1[47:32]
 IF (imm8[3] == 1) THEN DEST[63:48] \leftarrow SRC2[63:48]
 ELSE DEST[63:48] \leftarrow SRC1[63:48]
 IF (imm8[4] == 1) THEN DEST[79:64] \leftarrow SRC2[79:64]
 ELSE DEST[79:64] \leftarrow SRC1[79:64]
 IF (imm8[5] == 1) THEN DEST[95:80] \leftarrow SRC2[95:80]
 ELSE DEST[95:80] \leftarrow SRC1[95:80]
 IF (imm8[6] == 1) THEN DEST[111:96] \leftarrow SRC2[111:96]
 ELSE DEST[111:96] \leftarrow SRC1[111:96]
 IF (imm8[7] == 1) THEN DEST[127:112] \leftarrow SRC2[127:112]
 ELSE DEST[127:112] \leftarrow SRC1[127:112]
 IF (imm8[0] == 1) THEN DEST[143:128] \leftarrow SRC2[143:128]
 ELSE DEST[143:128] \leftarrow SRC1[143:128]
 IF (imm8[1] == 1) THEN DEST[159:144] \leftarrow SRC2[159:144]
 ELSE DEST[159:144] \leftarrow SRC1[159:144]
 IF (imm8[2] == 1) THEN DEST[175:160] \leftarrow SRC2[175:160]
 ELSE DEST[175:160] \leftarrow SRC1[175:160]
 IF (imm8[3] == 1) THEN DEST[191:176] \leftarrow SRC2[191:176]
 ELSE DEST[191:176] \leftarrow SRC1[191:176]
 IF (imm8[4] == 1) THEN DEST[207:192] \leftarrow SRC2[207:192]
 ELSE DEST[207:192] \leftarrow SRC1[207:192]
 IF (imm8[5] == 1) THEN DEST[223:208] \leftarrow SRC2[223:208]
 ELSE DEST[223:208] \leftarrow SRC1[223:208]
 IF (imm8[6] == 1) THEN DEST[239:224] \leftarrow SRC2[239:224]
 ELSE DEST[239:224] \leftarrow SRC1[239:224]
 IF (imm8[7] == 1) THEN DEST[255:240] \leftarrow SRC2[255:240]
 ELSE DEST[255:240] \leftarrow SRC1[255:240]

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW: `__m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);`

VPBLENDW: `__m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PCLMULQDQ - Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	PCLMUL- QDQ	Carry-less multiplication of one quadword of <i>xmm1</i> by one quadword of <i>xmm2/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm1</i> and <i>xmm2/m128</i> should be used.
VEX.NDS.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	Both PCL- MULQDQ and AVX flags	Carry-less multiplication of one quadword of <i>xmm2</i> by one quadword of <i>xmm3/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm2</i> and <i>xmm3/m128</i> should be used.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-10, other bits of the immediate byte are ignored.

Table 4-10. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2 ¹ [63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for Imm8.

Table 4-11. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ <i>xmm1</i> , <i>xmm2</i>	0000_0000B
PCLMULHQLQDQ <i>xmm1</i> , <i>xmm2</i>	0000_0001B
PCLMULLQHDDQ <i>xmm1</i> , <i>xmm2</i>	0001_0000B
PCLMULHQHDDQ <i>xmm1</i> , <i>xmm2</i>	0001_0001B

Operation**PCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
DEST[127] ← 0;
DEST[VLMAX-1:128] (Unmodified)

```

VPCLMULQDQ

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {

```



```

    TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[i - j])
  }
  DEST[i] ← TmpB[i];
}
DEST[VLMAX-1:127] ← 0;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ: `__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 74 /r ¹ PCMPEQB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed bytes in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 74 /r PCMPEQB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed bytes in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
0F 75 /r ¹ PCMPEQW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed words in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 75 /r PCMPEQW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed words in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
0F 76 /r ¹ PCMPEQD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed doublewords in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 76 /r PCMPEQD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed doublewords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed bytes in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed words in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed doublewords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed bytes in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed words in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed doublewords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PCMPEQB (with 64-bit operands)

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] = SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPEQB (with 128-bit operands)

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] = SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0], SRC2[127:0])
DEST[VLMAX-1:128] ← 0
```

VPCMPEQB (VEX.256 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_EQUAL(SRC1[255:128], SRC2[255:128])
```

PCMPEQW (with 64-bit operands)

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPEQW (with 128-bit operands)

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
```

```
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[127:112] = SRC[127:112]
    THEN DEST[127:112] ← FFFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

VPCMPEQW (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-1:128] ← 0
```

VPCMPEQW (VEX.256 encoded version)

```
DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
```

PCMPEQD (with 64-bit operands)

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
IF DEST[63:32] = SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;
```

PCMPEQD (with 128-bit operands)

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] = SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

VPCMPEQD (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-1:128] ← 0
```

VPCMPEQD (VEX.256 encoded version)

```
DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
```

Intel C/C++ Compiler Intrinsic Equivalents

```
PCMPEQB:    __m64 __mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW:    __m64 __mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD:    __m64 __mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB: __m128i __mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW: __m128i __mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD: __m128i __mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB:   __m256i __mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW:   __m256i __mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD:   __m256i __mm256_cmpeq_epi32 (__m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PCMPEQQ — Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed quadwords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed quadwords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
```

VPCMPEQQ (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPEQQ (VEX.256 encoded version)

```
DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_EQUAL(SRC1[255:128], SRC2[255:128])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)PCMPEQQ:   __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ:     __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 /r imm8 PCMPESTRI <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 61 /r ib VPCMPESTRI <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMP-ISTRM”), and generates an index stored to the count register (ECX/RCX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
 ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
 SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
 OFlag – IntRes2[0]
 AFlag – Reset
 PFlag – Reset

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	RCX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```
int __mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);
```


Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```

int  _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r imm8 PCMPESTRM <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in <i>XMM0</i>
VEX.128.66.0F3A.WIG 60 /r ib VPCMPESTRM <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in <i>XMM0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
 ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
 SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
 OFlag – IntRes2[0]
 AFlag – Reset
 PFlag – Reset

Note: In VEX.128 encoded versions, bits (VLMAX-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Length1	Length2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

```
__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);
```

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```
int  _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 64 /r ¹ PCMPGTB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed signed byte integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 64 /r PCMPGTB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
OF 65 /r ¹ PCMPGTW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed signed word integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 65 /r PCMPGTW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
OF 66 /r ¹ PCMPGTD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed signed doubleword integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 66 /r PCMPGTD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed doubleword integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
VEX.NDS.128.66.OF.WIG 64 /r VPCMPGTB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.OF.WIG 65 /r VPCMPGTW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.OF.WIG 66 /r VPCMPGTD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed doubleword integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.OF.WIG 64 /r VPCMPGTB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.OF.WIG 65 /r VPCMPGTW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.OF.WIG 66 /r VPCMPGTD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed doubleword integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source

operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PCMPGTB (with 64-bit operands)

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPGTB (with 128-bit operands)

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] > SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPGTB (VEX.256 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_GREATER(SRC1[255:128], SRC2[255:128])
```

PCMPGTW (with 64-bit operands)

```
IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] > SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPGTW (with 128-bit operands)

```

IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[63:48] > SRC[127:112]
    THEN DEST[127:112] ← FFFFH;
ELSE DEST[127:112] ← 0; FI;

```

VPCMPGTW (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

VPCMPGTW (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_GREATER(SRC1[255:128], SRC2[255:128])

```

PCMPGTD (with 64-bit operands)

```

IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; FI;
IF DEST[63:32] > SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
ELSE DEST[63:32] ← 0; FI;

```

PCMPGTD (with 128-bit operands)

```

IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] > SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
ELSE DEST[127:96] ← 0; FI;

```

VPCMPGTD (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

VPCMPGTD (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_GREATER(SRC1[255:128], SRC2[255:128])

```

Intel C/C++ Compiler Intrinsic Equivalents

```

PCMPGTB:    __m64 __mm_cmpgt_pi8 ( __m64 m1, __m64 m2)
PCMPGTW:    __m64 __mm_cmpgt_pi16 ( __m64 m1, __m64 m2)
DCMPGTD:    __m64 __mm_cmpgt_pi32 ( __m64 m1, __m64 m2)
(V)PCMPGTB: __m128i __mm_cmpgt_epi8 ( __m128i a, __m128i b)
(V)PCMPGTW: __m128i __mm_cmpgt_epi16 ( __m128i a, __m128i b)
(V)DCMPGTD: __m128i __mm_cmpgt_epi32 ( __m128i a, __m128i b)
VPCMPGTB:   __m256i __mm256_cmpgt_epi8 ( __m256i a, __m256i b)
VPCMPGTW:   __m256i __mm256_cmpgt_epi16 ( __m256i a, __m256i b)
VPCMPGTD:   __m256i __mm256_cmpgt_epi32 ( __m256i a, __m256i b)

```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PCMPGTQ — Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ <i>xmm1,xmm2/m128</i>	RM	V/V	SSE4_2	Compare packed signed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed qwords in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed qwords in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[63:0] > SRC[63:0])
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI
IF (DEST[127:64] > SRC[127:64])
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI
```

VPCMPGTQ (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPGTQ (VEX.256 encoded version)

```
DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_GREATER(SRC1[255:128], SRC2[255:128])
```


Intel C/C++ Compiler Intrinsic Equivalent

(V)PCMPGTQ: `__m128i _mm_cmpgt_epi64(__m128i a, __m128i b)`

VPCMPGTQ: `__m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 /r imm8 PCMPISTRI <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPSTRI / PCMPSTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
 ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
 SFlag – Set if any byte/word of xmm1 is null, reset otherwise
 OFlag –IntRes2[0]
 AFlag – Reset
 PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX
64 bit + REX.W	xmm	xmm/m128	RCX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```
int  _mm_cmpistri (__m128i a, __m128i b, const int mode);
```

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```
int  _mm_cmpistra (__m128i a, __m128i b, const int mode);
```

```
int  _mm_cmpistrb (__m128i a, __m128i b, const int mode);
```

```
int  _mm_cmpistro (__m128i a, __m128i b, const int mode);
```

```
int  _mm_cmpistrs (__m128i a, __m128i b, const int mode);
```

```
int  _mm_cmpistrz (__m128i a, __m128i b, const int mode);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 /r imm8 PCMPISTRM <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in <i>XMM0</i> .
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in <i>XMM0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”) generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
 ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
 SFlag – Set if any byte/word of xmm1 is null, reset otherwise
 OFlag – IntRes2[0]
 AFlag – Reset
 PFlag – Reset

Note: In VEX.128 encoded versions, bits (VLMAX-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0
64 bit + REX.W	xmm	xmm/m128	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

`__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode);`

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```

int  _mm_cmpistra (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int  _mm_cmpistro (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrz (__m128i a, __m128i b, const int mode);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PDEP — Parallel Bits Deposit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F2.0F38.W0 F5 /r PDEP <i>r32a</i> , <i>r32b</i> , <i>r/m32</i>	RVM	V/V	BMI2	Parallel deposit of bits from <i>r32b</i> using mask in <i>r/m32</i> , result is written to <i>r32a</i> .
VEX.NDS.LZ.F2.0F38.W1 F5 /r PDEP <i>r64a</i> , <i>r64b</i> , <i>r/m64</i>	RVM	V/N.E.	BMI2	Parallel deposit of bits from <i>r64b</i> using mask in <i>r/m64</i> , result is written to <i>r64a</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

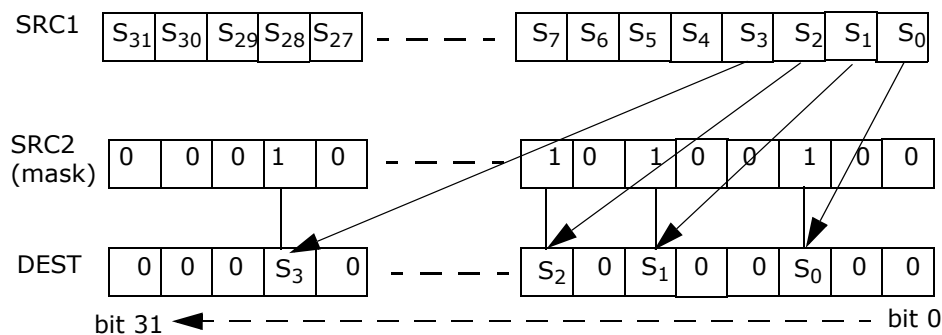


Figure 4-4. PDEP Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0;
m ← 0, k ← 0;
DO WHILE m < OperandSize
```

```
    IF MASK[m] = 1 THEN
        DEST[m] ← TEMP[k];
        k ← k + 1;
    FI
    m ← m + 1;
```

OD

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PDEP: unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);

PDEP: unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.

PEXT — Parallel Bits Extract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F5 /r PEXT <i>r32a</i> , <i>r32b</i> , <i>r/m32</i>	RVM	V/V	BMI2	Parallel extract of bits from <i>r32b</i> using mask in <i>r/m32</i> , result is written to <i>r32a</i> .
VEX.NDS.LZ.F3.0F38.W1 F5 /r PEXT <i>r64a</i> , <i>r64b</i> , <i>r/m64</i>	RVM	V/N.E.	BMI2	Parallel extract of bits from <i>r64b</i> using mask in <i>r/m64</i> , result is written to <i>r64a</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (<i>w</i>)	VEX.vvvv (<i>r</i>)	ModRM:r/m (<i>r</i>)	NA

Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

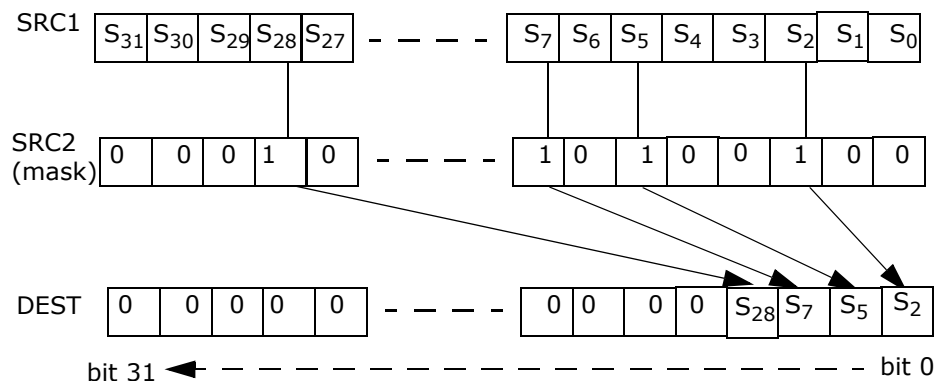


Figure 4-5. PEXT Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```

TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0;
m ← 0, k ← 0;
DO WHILE m < OperandSize

    IF MASK[ m ] = 1 THEN
        DEST[ k ] ← TEMP[ m ];
        k ← k + 1;
    m ← m + 1;

```



```
FI
m ← m+ 1;
```

OD

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
PEXT:      unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);
PEXT:      unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
66 0F 3A 16 /r ib PEXTRD <i>r/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r/m32</i> .
66 REX.W 0F 3A 16 /r ib PEXTRQ <i>r/m64, xmm2, imm8</i>	MRI	V/N.E.	SSE4_1	Extract a qword integer value from <i>xmm2</i> at the source qword offset specified by <i>imm8</i> into <i>r/m64</i> .
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V ¹ /V	AVX	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of <i>r64/r32</i> is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	MRI	V/V	AVX	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	MRI	V/i	AVX	Extract a qword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r64/m64</i> .

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from *imm8*[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros. Attempt to execute VPEXTRQ in non-64-bit mode will cause #UD.

Operation

CASE of

```

PEXTRB: SEL ← COUNT[3:0];
        TEMP ← (Src >> SEL*8) AND FFH;
        IF (DEST = Mem8)
            THEN
                Mem8 ← TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] ← TEMP[7:0];
                    r64[63:8] ← ZERO_FILL; ;
            ELSE
                R32[7:0] ← TEMP[7:0];
                r32[31:8] ← ZERO_FILL; ;
        FI;
PEXTRD: SEL ← COUNT[1:0];
        TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
        DEST ← TEMP;
PEXTRQ: SEL ← COUNT[0];
        TEMP ← (Src >> SEL*64);
        DEST ← TEMP;

```

EASC:

(V)PEXTRTD/(V)PEXTRQ

IF (64-Bit Mode and 64-bit dest operand)

THEN

```

Src_Offset ← Imm8[0]
r64/m64 ← (Src >> Src_Offset * 64)

```

ELSE

```

Src_Offset ← Imm8[1:0]
r32/m32 ← ((Src >> Src_Offset * 32) AND 0FFFFFFFh);

```

FI

(V)PEXTRB (dest=m8)

SRC_Offset ← Imm8[3:0]

Mem8 ← (Src >> SRC_Offset*8)

(V)PEXTRB (dest=reg)

IF (64-Bit Mode)

THEN

```

SRC_Offset ← Imm8[3:0]
DEST[7:0] ← ((Src >> SRC_Offset*8) AND 0FFh)
DEST[63:8] ← ZERO_FILL;

```

ELSE

```

SRC_Offset ← Imm8[3:0];
DEST[7:0] ← ((Src >> SRC_Offset*8) AND 0FFh);
DEST[31:8] ← ZERO_FILL;

```

FI

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB: int _mm_extract_epi8 (__m128i src, const int ndx);

PEXTRD: int _mm_extract_epi32 (__m128i src, const int ndx);

PEXTRQ: __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.
 If VPEXTRQ in non-64-bit mode, VEX.W=1.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C5 /r ib ¹ PEXTRW <i>reg, mm, imm8</i>	RMI	V/V	SSE	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>reg</i> , bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /r ib PEXTRW <i>reg, xmm, imm8</i>	RMI	V/V	SSE2	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>reg</i> , bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW <i>reg/m16, xmm, imm8</i>	MRI	V/V	SSE4_1	Extract the word specified by <i>imm8</i> from <i>xmm</i> and copy it to lowest 16 bits of <i>reg</i> or <i>m16</i> . Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	RMI	V ² /V	AVX	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F.3A.W0 15 /r ib VPEXTRW <i>reg/m16, xmm2, imm8</i>	MRI	V/V	AVX	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```

IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
THEN
    FOR (PEXTRW instruction with 64-bit source operand)
    { SEL ← COUNT[1:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r64[15:0] ← TEMP[15:0];
      r64[63:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
    { SEL ← COUNT[2:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r64[15:0] ← TEMP[15:0];
      r64[63:16] ← ZERO_FILL; }
ELSE
    FOR (PEXTRW instruction with 64-bit source operand)
    { SEL ← COUNT[1:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r32[15:0] ← TEMP[15:0];
      r32[31:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
    { SEL ← COUNT[2:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r32[15:0] ← TEMP[15:0];
      r32[31:16] ← ZERO_FILL; };
FI;
FI;

```

(V)PEXTRW (dest=m16)

```

SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> SRC_Offset*16)

```

(V)PEXTRW (dest=reg)

```

IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> SRC_Offset*16) AND 0FFFFh)
    DEST[63:16] ← ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> SRC_Offset*16) AND 0FFFFh)
    DEST[31:16] ← ZERO_FILL;
FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PEXTRW:    int _mm_extract_pi16 (__m64 a, int n)
PEXTRW:    int _mm_extract_epi16 (__m128i a, int imm)

```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PHADDW/PHADD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 01 /r ¹ PHADDW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to <i>mm1</i> .
66 0F 38 01 /r PHADDW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to <i>xmm1</i> .
0F 38 02 /r PHADD <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to <i>mm1</i> .
66 0F 38 02 /r PHADD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 02 /r VPHADD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 01 /r VPHADDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to <i>ymm1</i> .
VEX.NDS.256.66.0F38.WIG 02 /r VPHADD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

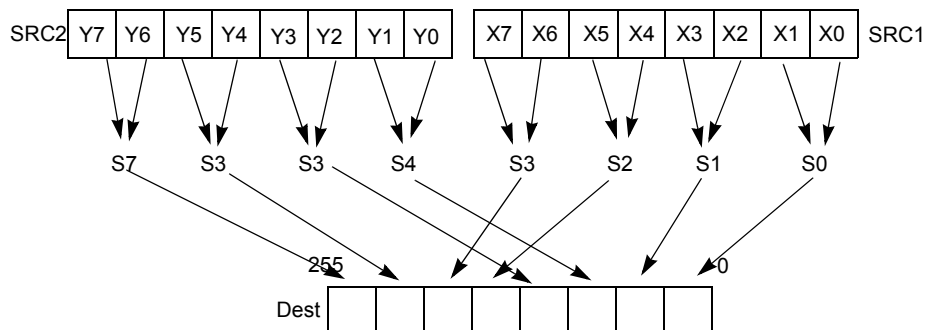


Figure 4-6. 256-bit VPHADD Instruction Operation

Operation

PHADDW (with 64-bit operands)

```
mm1[15:0] = mm1[31:16] + mm1[15:0];
mm1[31:16] = mm1[63:48] + mm1[47:32];
mm1[47:32] = mm2/m64[31:16] + mm2/m64[15:0];
mm1[63:48] = mm2/m64[63:48] + mm2/m64[47:32];
```

PHADDW (with 128-bit operands)

```
xmm1[15:0] = xmm1[31:16] + xmm1[15:0];
xmm1[31:16] = xmm1[63:48] + xmm1[47:32];
xmm1[47:32] = xmm1[95:80] + xmm1[79:64];
xmm1[63:48] = xmm1[127:112] + xmm1[111:96];
xmm1[79:64] = xmm2/m128[31:16] + xmm2/m128[15:0];
xmm1[95:80] = xmm2/m128[63:48] + xmm2/m128[47:32];
xmm1[111:96] = xmm2/m128[95:80] + xmm2/m128[79:64];
xmm1[127:112] = xmm2/m128[127:112] + xmm2/m128[111:96];
```

VPHADDW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[VLMAX-1:128] ← 0
```

VPHADDW (VEX.256 encoded version)

$\text{DEST}[15:0] \leftarrow \text{SRC1}[31:16] + \text{SRC1}[15:0]$
 $\text{DEST}[31:16] \leftarrow \text{SRC1}[63:48] + \text{SRC1}[47:32]$
 $\text{DEST}[47:32] \leftarrow \text{SRC1}[95:80] + \text{SRC1}[79:64]$
 $\text{DEST}[63:48] \leftarrow \text{SRC1}[127:112] + \text{SRC1}[111:96]$
 $\text{DEST}[79:64] \leftarrow \text{SRC2}[31:16] + \text{SRC2}[15:0]$
 $\text{DEST}[95:80] \leftarrow \text{SRC2}[63:48] + \text{SRC2}[47:32]$
 $\text{DEST}[111:96] \leftarrow \text{SRC2}[95:80] + \text{SRC2}[79:64]$
 $\text{DEST}[127:112] \leftarrow \text{SRC2}[127:112] + \text{SRC2}[111:96]$
 $\text{DEST}[143:128] \leftarrow \text{SRC1}[159:144] + \text{SRC1}[143:128]$
 $\text{DEST}[159:144] \leftarrow \text{SRC1}[191:176] + \text{SRC1}[175:160]$
 $\text{DEST}[175:160] \leftarrow \text{SRC1}[223:208] + \text{SRC1}[207:192]$
 $\text{DEST}[191:176] \leftarrow \text{SRC1}[255:240] + \text{SRC1}[239:224]$
 $\text{DEST}[207:192] \leftarrow \text{SRC2}[127:112] + \text{SRC2}[143:128]$
 $\text{DEST}[223:208] \leftarrow \text{SRC2}[159:144] + \text{SRC2}[175:160]$
 $\text{DEST}[239:224] \leftarrow \text{SRC2}[191:176] + \text{SRC2}[207:192]$
 $\text{DEST}[255:240] \leftarrow \text{SRC2}[223:208] + \text{SRC2}[239:224]$

PHADDD (with 64-bit operands)

$\text{mm1}[31:0] = \text{mm1}[63:32] + \text{mm1}[31:0];$
 $\text{mm1}[63:32] = \text{mm2}/\text{m64}[63:32] + \text{mm2}/\text{m64}[31:0];$

PHADDD (with 128-bit operands)

$\text{xmm1}[31:0] = \text{xmm1}[63:32] + \text{xmm1}[31:0];$
 $\text{xmm1}[63:32] = \text{xmm1}[127:96] + \text{xmm1}[95:64];$
 $\text{xmm1}[95:64] = \text{xmm2}/\text{m128}[63:32] + \text{xmm2}/\text{m128}[31:0];$
 $\text{xmm1}[127:96] = \text{xmm2}/\text{m128}[127:96] + \text{xmm2}/\text{m128}[95:64];$

VPHADDD (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPHADDD (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[191:160] + \text{SRC1}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[255:224] + \text{SRC1}[223:192]$
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[191:160] + \text{SRC2}[159:128]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[255:224] + \text{SRC2}[223:192]$

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW: `__m64 _mm_hadd_pi16 (__m64 a, __m64 b)`
 PHADDD: `__m64 _mm_hadd_pi32 (__m64 a, __m64 b)`
 (V)PHADDW: `__m128i _mm_hadd_epi16 (__m128i a, __m128i b)`
 (V)PHADDD: `__m128i _mm_hadd_epi32 (__m128i a, __m128i b)`
 VPHADDW: `__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)`
 VPHADDD: `__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHADDSW — Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 03 /r ¹ PHADDSW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to <i>mm1</i> .
66 OF 38 03 /r PHADDSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 03 /r VPHADDSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHADDSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord((mm1[31-16] + mm1[15-0]));
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

PHADDSW (with 128-bit operands)

```

xmm1[15:0] = SaturateToSignedWord(xmm1[31:16] + xmm1[15:0]);
xmm1[31:16] = SaturateToSignedWord(xmm1[63:48] + xmm1[47:32]);
xmm1[47:32] = SaturateToSignedWord(xmm1[95:80] + xmm1[79:64]);
xmm1[63:48] = SaturateToSignedWord(xmm1[127:112] + xmm1[111:96]);
xmm1[79:64] = SaturateToSignedWord(xmm2/m128[31:16] + xmm2/m128[15:0]);
xmm1[95:80] = SaturateToSignedWord(xmm2/m128[63:48] + xmm2/m128[47:32]);
xmm1[111:96] = SaturateToSignedWord(xmm2/m128[95:80] + xmm2/m128[79:64]);
xmm1[127:112] = SaturateToSignedWord(xmm2/m128[127:112] + xmm2/m128[111:96]);

```

VPHADDSW (VEX.128 encoded version)

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[VLMAX-1:128] ← 0

```

VPHADDSW (VEX.256 encoded version)

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128] = SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord(SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191:176] + SRC2[159:144])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PHADDSW:    __m64 _mm_hadds_pi16 (__m64 a, __m64 b)
(V)PHADDSW: __m128i _mm_hadds_epi16 (__m128i a, __m128i b)
VPHADDSW:   __m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHMINPOSUW — Packed Horizontal Word Minimum

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Find the minimum unsigned word in <i>xmm2/m128</i> and place its value in the low word of <i>xmm1</i> and its index in the second-lowest word of <i>xmm1</i> .
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Find the minimum unsigned word in <i>xmm2/m128</i> and place its value in the low word of <i>xmm1</i> and its index in the second-lowest word of <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHMINPOSUW (128-bit Legacy SSE version)

```

INDEX ← 0;
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN)
    THEN INDEX ← 1; MIN ← SRC[31:16]; FI;
IF (SRC[47:32] < MIN)
    THEN INDEX ← 2; MIN ← SRC[47:32]; FI;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
    THEN INDEX ← 7; MIN ← SRC[127:112]; FI;
DEST[15:0] ← MIN;
DEST[18:16] ← INDEX;
DEST[127:19] ← 00000000000000000000000000000000H;
```

VPHMINPOSUW (VEX.128 encoded version)INDEX \leftarrow 0MIN \leftarrow SRC[15:0]IF (SRC[31:16] < MIN) THEN INDEX \leftarrow 1; MIN \leftarrow SRC[31:16]IF (SRC[47:32] < MIN) THEN INDEX \leftarrow 2; MIN \leftarrow SRC[47:32]

* Repeat operation for words 3 through 6

IF (SRC[127:112] < MIN) THEN INDEX \leftarrow 7; MIN \leftarrow SRC[127:112]DEST[15:0] \leftarrow MINDEST[18:16] \leftarrow INDEXDEST[127:19] \leftarrow 00000000000000000000000000000000HDEST[VLMAX-1:128] \leftarrow 0**Intel C/C++ Compiler Intrinsic Equivalent**PHMINPOSUW: `__m128i _mm_minpos_epu16(__m128i packed_words);`**Flags Affected**

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv \neq 1111B.

PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 05 /r ¹ PHSUBW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to <i>mm1</i> .
66 0F 38 05 /r PHSUBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to <i>xmm1</i> .
0F 38 06 /r PHSUBD <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to <i>mm1</i> .
66 0F 38 06 /r PHSUBD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 05 /r VPHSUBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to <i>ymm1</i> .
VEX.NDS.256.66.0F38.WIG 06 /r VPHSUBD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBW (with 64-bit operands)

```
mm1[15:0] = mm1[15:0] - mm1[31:16];
mm1[31:16] = mm1[47:32] - mm1[63:48];
mm1[47:32] = mm2/m64[15:0] - mm2/m64[31:16];
mm1[63:48] = mm2/m64[47:32] - mm2/m64[63:48];
```

PHSUBW (with 128-bit operands)

```
xmm1[15:0] = xmm1[15:0] - xmm1[31:16];
xmm1[31:16] = xmm1[47:32] - xmm1[63:48];
xmm1[47:32] = xmm1[79:64] - xmm1[95:80];
xmm1[63:48] = xmm1[111:96] - xmm1[127:112];
xmm1[79:64] = xmm2/m128[15:0] - xmm2/m128[31:16];
xmm1[95:80] = xmm2/m128[47:32] - xmm2/m128[63:48];
xmm1[111:96] = xmm2/m128[79:64] - xmm2/m128[95:80];
xmm1[127:112] = xmm2/m128[111:96] - xmm2/m128[127:112];
```

VPHSUBW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[VLMAX-1:128] ← 0
```

VPHSUBW (VEX.256 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]
```

PHSUBD (with 64-bit operands)

```
mm1[31:0] = mm1[31:0] - mm1[63:32];
mm1[63:32] = mm2/m64[31:0] - mm2/m64[63:32];
```

PHSUBD (with 128-bit operands)

$\text{xmm1}[31:0] = \text{xmm1}[31:0] - \text{xmm1}[63:32];$
 $\text{xmm1}[63:32] = \text{xmm1}[95:64] - \text{xmm1}[127:96];$
 $\text{xmm1}[95:64] = \text{xmm2}/\text{m128}[31:0] - \text{xmm2}/\text{m128}[63:32];$
 $\text{xmm1}[127:96] = \text{xmm2}/\text{m128}[95:64] - \text{xmm2}/\text{m128}[127:96];$

VPHSUBD (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPHSUBD (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224]$
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[159:128] - \text{SRC2}[191:160]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[223:192] - \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW: `__m64 _mm_hsub_pi16 (__m64 a, __m64 b)`
 PHSUBD: `__m64 _mm_hsub_pi32 (__m64 a, __m64 b)`
 (V)PHSUBW: `__m128i _mm_hsub_epi16 (__m128i a, __m128i b)`
 (V)PHSUBD: `__m128i _mm_hsub_epi32 (__m128i a, __m128i b)`
 VPHSUBW: `__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)`
 VPHSUBD: `__m256i _mm256_hsub_epi32 (__m256i a, __m256i b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHSUBSW — Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 07 /r ¹ PHSUBSW mm1, mm2/m64	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to mm1.
66 0F 38 07 /r PHSUBSW xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.NDS.256.66.0F38.WIG 07 /r VPHSUBSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to ymm1.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

PHSUBSW (with 128-bit operands)

```

xmm1[15:0] = SaturateToSignedWord(xmm1[15:0] - xmm1[31:16]);
xmm1[31:16] = SaturateToSignedWord(xmm1[47:32] - xmm1[63:48]);
xmm1[47:32] = SaturateToSignedWord(xmm1[79:64] - xmm1[95:80]);
xmm1[63:48] = SaturateToSignedWord(xmm1[111:96] - xmm1[127:112]);
xmm1[79:64] = SaturateToSignedWord(xmm2/m128[15:0] - xmm2/m128[31:16]);
xmm1[95:80] = SaturateToSignedWord(xmm2/m128[47:32] - xmm2/m128[63:48]);
xmm1[111:96] = SaturateToSignedWord(xmm2/m128[79:64] - xmm2/m128[95:80]);
xmm1[127:112] = SaturateToSignedWord(xmm2/m128[111:96] - xmm2/m128[127:112]);

```

VPHSUBSW (VEX.128 encoded version)

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[VLMAX-1:128] ← 0

```

VPHSUBSW (VEX.256 encoded version)

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128] = SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PHSUBSW:      __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
(V)PHSUBSW:   __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
VPHSUBSW:     __m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 3A 20 /r ib PINSRB <i>xmm1</i> , <i>r32/m8</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Insert a byte integer value from <i>r32/m8</i> into <i>xmm1</i> at the destination element in <i>xmm1</i> specified by <i>imm8</i> .
66 OF 3A 22 /r ib PINSRD <i>xmm1</i> , <i>r/m32</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Insert a dword integer value from <i>r/m32</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
66 REX.W OF 3A 22 /r ib PINSRQ <i>xmm1</i> , <i>r/m64</i> , <i>imm8</i>	RMI	V/N. E.	SSE4_1	Insert a qword integer value from <i>r/m64</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB <i>xmm1</i> , <i>xmm2</i> , <i>r32/m8</i> , <i>imm8</i>	RVMI	V ¹ /V	AVX	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1</i> , <i>xmm2</i> , <i>r/m32</i> , <i>imm8</i>	RVMI	V/V	AVX	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1</i> , <i>xmm2</i> , <i>r/m64</i> , <i>imm8</i>	RVMI	V/I	AVX	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

Operation

CASE OF

```

PINSRB: SEL ← COUNT[3:0];
        MASK ← (OFFH << (SEL * 8));
        TEMP ← (((SRC[7:0] << (SEL * 8)) AND MASK);
PINSRD: SEL ← COUNT[1:0];
        MASK ← (OFFFFFFFFFH << (SEL * 32));
        TEMP ← (((SRC << (SEL * 32)) AND MASK) ;
PINSRQ: SEL ← COUNT[0]
        MASK ← (OFFFFFFFFFFFFFFFFFFFH << (SEL * 64));
        TEMP ← (((SRC << (SEL * 32)) AND MASK) ;

```

ESAC;

DEST ← ((DEST AND NOT MASK) OR TEMP);

VPINSRB (VEX.128 encoded version)

```

SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

VPINSRD (VEX.128 encoded version)

```

SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

VPINSRQ (VEX.128 encoded version)

```

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PINSRB:    __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD:    __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ:    __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

```

#UD          If VEX.L = 1.
              If VPINSRQ in non-64-bit mode with VEX.W=1.

```

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C4 /r ib ¹ PINSRW <i>mm</i> , <i>r32/m16</i> , <i>imm8</i>	RMI	V/V	SSE	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i> .
66 0F C4 /r ib PINSRW <i>xmm</i> , <i>r32/m16</i> , <i>imm8</i>	RMI	V/V	SSE2	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW <i>xmm1</i> , <i>xmm2</i> , <i>r32/m16</i> , <i>imm8</i>	RVMI	V ² /V	AVX	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .

NOTES:

- See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.
- In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PINSRW (with 64-bit source operand)

SEL ← COUNT AND 3H;

CASE (Determine word position) OF

SEL ← 0: MASK ← 000000000000FFFFH;

SEL ← 1: MASK ← 00000000FFFF0000H;

SEL ← 2: MASK ← 0000FFFF00000000H;

SEL ← 3: MASK ← FFFF000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);

PINSRW (with 128-bit source operand)

SEL ← COUNT AND 7H;

CASE (Determine word position) OF

SEL ← 0: MASK ← 0000000000000000000000000000FFFFH;

SEL ← 1: MASK ← 000000000000000000000000FFFF0000H;

SEL ← 2: MASK ← 0000000000000000000000FFFF00000000H;

SEL ← 3: MASK ← 0000000000000000FFFF000000000000H;

SEL ← 4: MASK ← 000000000000FFFF00000000000000000H;

SEL ← 5: MASK ← 00000000FFFF000000000000000000000H;

SEL ← 6: MASK ← 0000FFFF000000000000000000000000H;

SEL ← 7: MASK ← FFFF0000000000000000000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);

VPINSRW (VEX.128 encoded version)

SEL ← imm8[2:0]

DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW: __m64 _mm_insert_pi16 (__m64 a, int d, int n)

PINSRW: __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.

If VPINSRW in non-64-bit mode with VEX.W=1.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 04 /r ¹ PMADDUBSW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>mm1</i> .
66 0F 38 04 /r PMADDUBSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMADDUBSW (with 64 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (with 128 bit operands)

```
DEST[15:0] = SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127:112] = SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]* DEST[119:112]);
```

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[VLMAX-1:128] ← 0
```

VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
```

Intel C/C++ Compiler Intrinsic Equivalents

```
PMADDUBSW:    __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW: __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW:   __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F5 /r ¹ PMADDWD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Multiply the packed words in <i>mm</i> by the packed words in <i>mm/m64</i> , add adjacent doubleword results, and store in <i>mm</i> .
66 0F F5 /r PMADDWD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed word integers in <i>xmm1</i> by the packed word integers in <i>xmm2/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed word integers in <i>xmm2</i> by the packed word integers in <i>xmm3/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed word integers in <i>ymm2</i> by the packed word integers in <i>ymm3/m256</i> , add adjacent doubleword results, and store in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-7 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

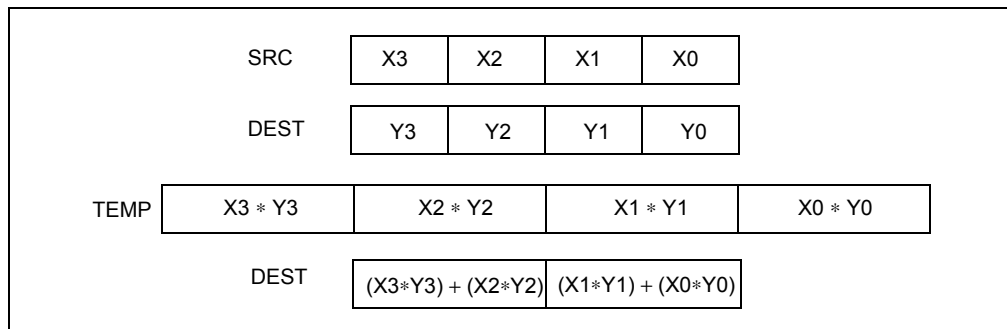


Figure 4-7. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (with 64-bit operands)

$$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]);$$

$$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]);$$

PMADDWD (with 128-bit operands)

$$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]);$$

$$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]);$$

$$\text{DEST}[95:64] \leftarrow (\text{DEST}[79:64] * \text{SRC}[79:64]) + (\text{DEST}[95:80] * \text{SRC}[95:80]);$$

$$\text{DEST}[127:96] \leftarrow (\text{DEST}[111:96] * \text{SRC}[111:96]) + (\text{DEST}[127:112] * \text{SRC}[127:112]);$$

VPMADDWD (VEX.128 encoded version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[15:0] * \text{SRC2}[15:0]) + (\text{SRC1}[31:16] * \text{SRC2}[31:16])$$

$$\text{DEST}[63:32] \leftarrow (\text{SRC1}[47:32] * \text{SRC2}[47:32]) + (\text{SRC1}[63:48] * \text{SRC2}[63:48])$$

$$\text{DEST}[95:64] \leftarrow (\text{SRC1}[79:64] * \text{SRC2}[79:64]) + (\text{SRC1}[95:80] * \text{SRC2}[95:80])$$

$$\text{DEST}[127:96] \leftarrow (\text{SRC1}[111:96] * \text{SRC2}[111:96]) + (\text{SRC1}[127:112] * \text{SRC2}[127:112])$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$

VPMADDWD (VEX.256 encoded version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[15:0] * \text{SRC2}[15:0]) + (\text{SRC1}[31:16] * \text{SRC2}[31:16])$$

$$\text{DEST}[63:32] \leftarrow (\text{SRC1}[47:32] * \text{SRC2}[47:32]) + (\text{SRC1}[63:48] * \text{SRC2}[63:48])$$

$$\text{DEST}[95:64] \leftarrow (\text{SRC1}[79:64] * \text{SRC2}[79:64]) + (\text{SRC1}[95:80] * \text{SRC2}[95:80])$$

$$\text{DEST}[127:96] \leftarrow (\text{SRC1}[111:96] * \text{SRC2}[111:96]) + (\text{SRC1}[127:112] * \text{SRC2}[127:112])$$

$$\text{DEST}[159:128] \leftarrow (\text{SRC1}[143:128] * \text{SRC2}[143:128]) + (\text{SRC1}[159:144] * \text{SRC2}[159:144])$$

$$\text{DEST}[191:160] \leftarrow (\text{SRC1}[175:160] * \text{SRC2}[175:160]) + (\text{SRC1}[191:176] * \text{SRC2}[191:176])$$

$$\text{DEST}[223:192] \leftarrow (\text{SRC1}[207:192] * \text{SRC2}[207:192]) + (\text{SRC1}[223:208] * \text{SRC2}[223:208])$$

$$\text{DEST}[255:224] \leftarrow (\text{SRC1}[239:224] * \text{SRC2}[239:224]) + (\text{SRC1}[255:240] * \text{SRC2}[255:240])$$

Intel C/C++ Compiler Intrinsic Equivalent

PMADDWD: `__m64 _mm_madd_pi16(__m64 m1, __m64 m2)`
 (V)PMADDWD: `__m128i _mm_madd_epi16 (__m128i a, __m128i b)`
 VPMADDWD: `__m256i _mm256_madd_epi16 (__m256i a, __m256i b)`

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PMAXSB — Maximum of Packed Signed Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3C /r PMAXSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m128</i> and store packed maximum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[7:0] > SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; FI;
IF (DEST[15:8] > SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; FI;
IF (DEST[23:16] > SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; FI;
IF (DEST[31:24] > SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; FI;
IF (DEST[39:32] > SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; FI;
IF (DEST[47:40] > SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];

```

```

    ELSE DEST[47:40] ← SRC[47:40]; FI;
IF (DEST[55:48] > SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; FI;
IF (DEST[63:56] > SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; FI;
IF (DEST[71:64] > SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; FI;
IF (DEST[79:72] > SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; FI;
IF (DEST[87:80] > SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; FI;
IF (DEST[95:88] > SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; FI;
IF (DEST[103:96] > SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; FI;
IF (DEST[111:104] > SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; FI;
IF (DEST[119:112] > SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; FI;
IF (DEST[127:120] > SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; FI;

```

VPMASB (VEX.128 encoded version)

```

IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMASB (VEX.256 encoded version)

```

IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] > SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMAXSB: `__m128i _mm_max_epi8 (__m128i a, __m128i b);`

VPMAXSB: `__m256i _mm256_max_epi8 (__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXSD — Maximum of Packed Signed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3D /r PMAXSD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed dword integers in <i>ymm2</i> and <i>ymm3/m128</i> and store packed maximum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;

```

VPMAXSD (VEX.128 encoded version)

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE

```

```

    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] > SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMAXSD (VEX.256 encoded version)

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PMAXSD:    __m128i _mm_max_epi32 ( __m128i a, __m128i b);
VPMAXSD:   __m256i _mm256_max_epi32 ( __m256i a, __m256i b);

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXSW—Maximum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EE /r ¹ PMAXSW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSE	Compare signed word integers in <i>mm2/m64</i> and <i>mm1</i> and return maximum values.
66 0F EE /r PMAXSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare signed word integers in <i>xmm2/m128</i> and <i>xmm1</i> and return maximum values.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed word integers in <i>xmm3/m128</i> and <i>xmm2</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed word integers in <i>ymm3/m128</i> and <i>ymm2</i> and store packed maximum values in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of word integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXSW (64-bit operands)

IF DEST[15:0] > SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(* Repeat operation for 2nd and 3rd words in source and destination operands *)

```

IF DEST[63:48] > SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;

```

PMAWSW (128-bit operands)

```

IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] > SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;

```

VPMAWSW (VEX.128 encoded version)

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] > SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMAWSW (VEX.256 encoded version)

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] > SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PMAWSW:    __m64 _mm_max_pi16(__m64 a, __m64 b)
(V)PMAWSW: __m128i _mm_max_epi16 (__m128i a, __m128i b)
VPMAWSW:   __m256i _mm256_max_epi16 (__m256i a, __m256i b)

```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF DE /r ¹ PMAXUB mm1, mm2/m64	RM	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.
66 OF DE /r PMAXUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare unsigned byte integers in xmm2/m128 and xmm1 and returns maximum values.
VEX.NDS.128.66.OF.WIG DE /r VPMAXUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.OF.WIG DE /r VPMAXUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of byte integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXUB (64-bit operands)

```
IF DEST[7:0] > SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
```

(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
 IF DEST[63:56] > SRC[63:56] THEN
 DEST[63:56] ← DEST[63:56];
 ELSE
 DEST[63:56] ← SRC[63:56]; FI;

PMAXUB (128-bit operands)

IF DEST[7:0] > SRC[7:0] THEN
 DEST[7:0] ← DEST[7:0];
 ELSE
 DEST[7:0] ← SRC[7:0]; FI;
 (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
 IF DEST[127:120] > SRC[127:120] THEN
 DEST[127:120] ← DEST[127:120];
 ELSE
 DEST[127:120] ← SRC[127:120]; FI;

VPMAXUB (VEX.128 encoded version)

IF SRC1[7:0] > SRC2[7:0] THEN
 DEST[7:0] ← SRC1[7:0];
 ELSE
 DEST[7:0] ← SRC2[7:0]; FI;
 (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
 IF SRC1[127:120] > SRC2[127:120] THEN
 DEST[127:120] ← SRC1[127:120];
 ELSE
 DEST[127:120] ← SRC2[127:120]; FI;
 DEST[VLMAX-1:128] ← 0

VPMAXUB (VEX.256 encoded version)

IF SRC1[7:0] > SRC2[7:0] THEN
 DEST[7:0] ← SRC1[7:0];
 ELSE
 DEST[15:0] ← SRC2[7:0]; FI;
 (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
 IF SRC1[255:248] > SRC2[255:248] THEN
 DEST[255:248] ← SRC1[255:248];
 ELSE
 DEST[255:248] ← SRC2[255:248]; FI;

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB: __m64 _mm_max_pu8(__m64 a, __m64 b)
 (V)PMAXUB: __m128i _mm_max_epu8 (__m128i a, __m128i b)
 VPMAXUB: __m256i _mm256_max_epu8 (__m256i a, __m256i b);

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXUD — Maximum of Packed Unsigned Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned dword integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;

```

VPMAXUD (VEX.128 encoded version)

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE

```

```

    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] > SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMAXUD (VEX.256 encoded version)

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PMAXUD: __m128i _mm_max_epu32 ( __m128i a, __m128i b);
VPMAXUD:   __m256i _mm256_max_epu32 ( __m256i a, __m256i b);

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXUW — Maximum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3E /r PMAXUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3E/r VPMAXUW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and store maximum packed values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and store maximum packed values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[15:0] > SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; FI;
IF (DEST[31:16] > SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; FI;
IF (DEST[47:32] > SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; FI;
IF (DEST[63:48] > SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; FI;
IF (DEST[79:64] > SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; FI;
IF (DEST[95:80] > SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];

```

```

    ELSE DEST[95:80] ← SRC[95:80]; FI;
IF (DEST[111:96] > SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; FI;
IF (DEST[127:112] > SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; FI;

```

VPMAXUW (VEX.128 encoded version)

```

    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] > SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMAXUW (VEX.256 encoded version)

```

    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] > SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PMAXUW:  __m128i _mm_max_epu16 ( __m128i a, __m128i b);
VPMAXUW:    __m256i _mm256_max_epu16 ( __m256i a, __m256i b)

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINSB — Minimum of Packed Signed Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 38 /r PMINSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 38 /r VPMINSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 38 /r VPMINSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[7:0] < SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; FI;
IF (DEST[15:8] < SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; FI;
IF (DEST[23:16] < SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; FI;
IF (DEST[31:24] < SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; FI;
IF (DEST[39:32] < SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; FI;
IF (DEST[47:40] < SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];

```

```

    ELSE DEST[47:40] ← SRC[47:40]; FI;
IF (DEST[55:48] < SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; FI;
IF (DEST[63:56] < SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; FI;
IF (DEST[71:64] < SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; FI;
IF (DEST[79:72] < SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; FI;
IF (DEST[87:80] < SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; FI;
IF (DEST[95:88] < SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; FI;
IF (DEST[103:96] < SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; FI;
IF (DEST[111:104] < SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; FI;
IF (DEST[119:112] < SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; FI;
IF (DEST[127:120] < SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; FI;

```

VPMINSB (VEX.128 encoded version)

```

IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMINSB (VEX.256 encoded version)

```

IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] < SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINSB: __m128i _mm_min_epi8 (__m128i a, __m128i b);
VPMINSB: __m256i _mm256_min_epi8 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PMINSD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed signed dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed dword integers in <i>ymm2</i> and <i>ymm3/m128</i> and store packed minimum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;

```

VPMINSD (VEX.128 encoded version)

```

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE

```



```

    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] < SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMINSD (VEX.256 encoded version)

```

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PMINSD:    __m128i _mm_min_epi32 ( __m128i a, __m128i b);
VPMINSD:      __m256i _mm256_min_epi32 (__m256i a, __m256i b);

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINSW—Minimum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EA /r ¹ PMINSW mm1, mm2/m64	RM	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 0F EA /r PMINSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare signed word integers in xmm2/m128 and xmm1 and return minimum values.
VEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of word integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMINSW (64-bit operands)

IF DEST[15:0] < SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(* Repeat operation for 2nd and 3rd words in source and destination operands *)

```

IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;

```

PMINSW (128-bit operands)

```

IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC/m64[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;

```

VPMSW (VEX.128 encoded version)

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] < SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMSW (VEX.256 encoded version)

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

PMINSW:      __m64 _mm_min_pi16 (__m64 a, __m64 b)
(V)PMINSW:   __m128i _mm_min_epi16 (__m128i a, __m128i b)
VPMSW:       __m256i _mm256_min_epi16 (__m256i a, __m256i b)

```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DA /r ¹ PMINUB mm1, mm2/m64	RM	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare unsigned byte integers in xmm2/m128 and xmm1 and returns minimum values.
VEX.NDS.128.66.0F.WIG DA /r VPMINUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F.WIG DA /r VPMINUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of byte integers to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMINUB (for 64-bit operands)

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
```

(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
 IF DEST[63:56] < SRC[63:56] THEN
 DEST[63:56] ← DEST[63:56];
 ELSE
 DEST[63:56] ← SRC[63:56]; FI;

PMINUB (for 128-bit operands)

IF DEST[7:0] < SRC[7:0] THEN
 DEST[7:0] ← DEST[7:0];
 ELSE
 DEST[7:0] ← SRC[7:0]; FI;
 (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
 IF DEST[127:120] < SRC[127:120] THEN
 DEST[127:120] ← DEST[127:120];
 ELSE
 DEST[127:120] ← SRC[127:120]; FI;

VPMINUB (VEX.128 encoded version)

VPMINUB instruction for 128-bit operands:

IF SRC1[7:0] < SRC2[7:0] THEN
 DEST[7:0] ← SRC1[7:0];
 ELSE
 DEST[7:0] ← SRC2[7:0]; FI;
 (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
 IF SRC1[127:120] < SRC2[127:120] THEN
 DEST[127:120] ← SRC1[127:120];
 ELSE
 DEST[127:120] ← SRC2[127:120]; FI;
 DEST[VLMAX-1:128] ← 0

VPMINUB (VEX.256 encoded version)

VPMINUB instruction for 128-bit operands:

IF SRC1[7:0] < SRC2[7:0] THEN
 DEST[7:0] ← SRC1[7:0];
 ELSE
 DEST[15:0] ← SRC2[7:0]; FI;
 (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
 IF SRC1[255:248] < SRC2[255:248] THEN
 DEST[255:248] ← SRC1[255:248];
 ELSE
 DEST[255:248] ← SRC2[255:248]; FI;

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB: __m64 _m_min_pu8 (__m64 a, __m64 b)
 (V)PMINUB: __m128i _mm_min_epu8 (__m128i a, __m128i b)
 VPMINUB: __m256i _mm256_min_epu8 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PMINUD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned dword integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned dword integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;

```

VPMINUD (VEX.128 encoded version)

VPMINUD instruction for 128-bit operands:

```

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];

```



```

ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:95] < SRC2[127:95] THEN
    DEST[127:95] ← SRC1[127:95];
ELSE
    DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMINUD (VEX.256 encoded version)

VPMINUD instruction for 128-bit operands:

```

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINUD: `__m128i _mm_min_epu32 (__m128i a, __m128i b);`

VPMINUD: `__m256i _mm256_min_epu32 (__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINUW — Minimum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3A /r PMINUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 3A/r VPMINUW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and return packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 3A /r VPMINUW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and return packed minimum values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```

IF (DEST[15:0] < SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; FI;
IF (DEST[31:16] < SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; FI;
IF (DEST[47:32] < SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; FI;
IF (DEST[63:48] < SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; FI;
IF (DEST[79:64] < SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; FI;
IF (DEST[95:80] < SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];

```

```

ELSE DEST[95:80] ← SRC[95:80]; FI;
IF (DEST[111:96] < SRC[111:96])
  THEN DEST[111:96] ← DEST[111:96];
  ELSE DEST[111:96] ← SRC[111:96]; FI;
IF (DEST[127:112] < SRC[127:112])
  THEN DEST[127:112] ← DEST[127:112];
  ELSE DEST[127:112] ← SRC[127:112]; FI;

```

VPMINUW (VEX.128 encoded version)

VPMINUW instruction for 128-bit operands:

```

IF SRC1[15:0] < SRC2[15:0] THEN
  DEST[15:0] ← SRC1[15:0];
ELSE
  DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] < SRC2[127:112] THEN
  DEST[127:112] ← SRC1[127:112];
ELSE
  DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX-1:128] ← 0

```

VPMINUW (VEX.256 encoded version)

VPMINUW instruction for 128-bit operands:

```

IF SRC1[15:0] < SRC2[15:0] THEN
  DEST[15:0] ← SRC1[15:0];
ELSE
  DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
  DEST[255:240] ← SRC1[255:240];
ELSE
  DEST[255:240] ← SRC2[255:240]; FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINUW: `__m128i _mm_min_epu16 (__m128i a, __m128i b);`

VPMINUW: `__m256i _mm256_min_epu16 (__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMOVMASKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D7 /r ¹ PMOVMASKB <i>reg, mm</i>	RM	V/V	SSE	Move a byte mask of <i>mm</i> to <i>reg</i> . The upper bits of r32 or r64 are zeroed
66 0F D7 /r PMOVMASKB <i>reg, xmm</i>	RM	V/V	SSE2	Move a byte mask of <i>xmm</i> to <i>reg</i> . The upper bits of r32 or r64 are zeroed
VEX.128.66.0F.WIG D7 /r VPMOVMASKB <i>reg, xmm1</i>	RM	V/V	AVX	Move a byte mask of <i>xmm1</i> to <i>reg</i> . The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMASKB <i>reg, ymm1</i>	RM	V/V	AVX2	Move a 32-bit mask of <i>ymm1</i> to <i>reg</i> . The upper bits of r64 are filled with zeros.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMASKB (with 64-bit source operand and r32)

```

r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;

```

(V)PMOVMASKB (with 128-bit source operand and r32)

```

r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;

```

VPMOVMASKB (with 256-bit source operand and r32)

$$r32[0] \leftarrow SRC[7];$$

$$r32[1] \leftarrow SRC[15];$$

(* Repeat operation for bytes 3rd through 31*)

$$r32[31] \leftarrow SRC[255];$$
PMOVMASKB (with 64-bit source operand and r64)

$$r64[0] \leftarrow SRC[7];$$

$$r64[1] \leftarrow SRC[15];$$

(* Repeat operation for bytes 2 through 6 *)

$$r64[7] \leftarrow SRC[63];$$

$$r64[63:8] \leftarrow ZERO_FILL;$$
(V)PMOVMASKB (with 128-bit source operand and r64)

$$r64[0] \leftarrow SRC[7];$$

$$r64[1] \leftarrow SRC[15];$$

(* Repeat operation for bytes 2 through 14 *)

$$r64[15] \leftarrow SRC[127];$$

$$r64[63:16] \leftarrow ZERO_FILL;$$
VPMOVMASKB (with 256-bit source operand and r64)

$$r64[0] \leftarrow SRC[7];$$

$$r64[1] \leftarrow SRC[15];$$

(* Repeat operation for bytes 2 through 31*)

$$r64[31] \leftarrow SRC[255];$$

$$r64[63:32] \leftarrow ZERO_FILL;$$
Intel C/C++ Compiler Intrinsic Equivalent

PMOVMASKB: `int _mm_movemask_pi8(__m64 a)`

(V)PMOVMASKB: `int _mm_movemask_epi8 (__m128i a)`

VPMOVMASKB: `int _mm256_movemask_epi8 (__m256i a)`

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv \neq 1111B.

PMOVSX — Packed Move with Sign Extend

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE4_1	Sign extend 8 packed signed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed signed 16-bit integers in <i>xmm1</i> .
66 0f 38 21 /r PMOVSXBD <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE4_1	Sign extend 4 packed signed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed signed 32-bit integers in <i>xmm1</i> .
66 0f 38 22 /r PMOVSXBQ <i>xmm1</i> , <i>xmm2/m16</i>	RM	V/V	SSE4_1	Sign extend 2 packed signed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed signed 64-bit integers in <i>xmm1</i> .
66 0f 38 23 /r PMOVSXWD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE4_1	Sign extend 4 packed signed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed signed 32-bit integers in <i>xmm1</i> .
66 0f 38 24 /r PMOVSXWQ <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE4_1	Sign extend 2 packed signed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed signed 64-bit integers in <i>xmm1</i> .
66 0f 38 25 /r PMOVSXDQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE4_1	Sign extend 2 packed signed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed signed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 16-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ <i>xmm1</i> , <i>xmm2/m16</i>	RM	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW <i>ymm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX2	Sign extend 16 packed 8-bit integers in <i>xmm2/m128</i> to 16 packed 16-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD <i>ymm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 32-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ <i>ymm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD <i>ymm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 8 packed 32-bit integers in <i>ymm1</i> .

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 25 /r VPMOVSDQ <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 4 packed 64-bit integers in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sign-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination register is YMM Register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMOVSXBW

```
DEST[15:0] ← SignExtend(SRC[7:0]);
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);
```

PMOVSXBD

```
DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);
```

PMOVSXBQ

```
DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);
```

PMOVSXWD

```
DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);
```

PMOVSXWQ

DEST[63:0] \leftarrow SignExtend(SRC[15:0]);
 DEST[127:64] \leftarrow SignExtend(SRC[31:16]);

PMOVSXDQ

DEST[63:0] \leftarrow SignExtend(SRC[31:0]);
 DEST[127:64] \leftarrow SignExtend(SRC[63:32]);

VPMOVSXBW (VEX.128 encoded version)

Packed_Sign_Extend_BYTE_to_WORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVSXBD (VEX.128 encoded version)

Packed_Sign_Extend_BYTE_to_DWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVSXBQ (VEX.128 encoded version)

Packed_Sign_Extend_BYTE_to_QWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVSXWD (VEX.128 encoded version)

Packed_Sign_Extend_WORD_to_DWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVSXWQ (VEX.128 encoded version)

Packed_Sign_Extend_WORD_to_QWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVSXDQ (VEX.128 encoded version)

Packed_Sign_Extend_DWORD_to_QWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVSXBW (VEX.256 encoded version)

Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
 Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])

VPMOVSXBD (VEX.256 encoded version)

Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
 Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])

VPMOVSXBQ (VEX.256 encoded version)

Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
 Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])

VPMOVSXWD (VEX.256 encoded version)

Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
 Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])

VPMOVSXWQ (VEX.256 encoded version)

Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
 Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])

VPMOVSXDQ (VEX.256 encoded version)

Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])

Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PMOVSXBW:    __m128i _mm_cvtepi8_epi16 ( __m128i a);
VPMOVSXBW:      __m256i _mm256_cvtepi8_epi16 ( __m128i a);
(V)PMOVSXBD:    __m128i _mm_cvtepi8_epi32 ( __m128i a);
VPMOVSXBD:      __m256i _mm256_cvtepi8_epi32 ( __m128i a);
(V)PMOVSXBQ:    __m128i _mm_cvtepi8_epi64 ( __m128i a);
VPMOVSXBQ:      __m256i _mm256_cvtepi8_epi64 ( __m128i a);
(V)PMOVSXWD:    __m128i _mm_cvtepi16_epi32 ( __m128i a);
VPMOVSXWD:      __m256i _mm256_cvtepi16_epi32 ( __m128i a);
(V)PMOVSXWQ:    __m128i _mm_cvtepi16_epi64 ( __m128i a);
VPMOVSXWQ:      __m256i _mm256_cvtepi16_epi64 ( __m128i a);
(V)PMOVSXDQ:    __m128i _mm_cvtepi32_epi64 ( __m128i a);
VPMOVSXDQ:      __m256i _mm256_cvtepi32_epi64 ( __m128i a);

```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

```

#UD              If VEX.L = 1.
                  If VEX.vvvv ≠ 1111B.

```

PMOVZX — Packed Move with Zero Extend

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 16-bit integers in <i>xmm1</i> .
66 0f 38 31 /r PMOVZXBW <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 32-bit integers in <i>xmm1</i> .
66 0f 38 32 /r PMOVZXBQ <i>xmm1</i> , <i>xmm2/m16</i>	RM	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed 64-bit integers in <i>xmm1</i> .
66 0f 38 33 /r PMOVZXWD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 32-bit integers in <i>xmm1</i> .
66 0f 38 34 /r PMOVZXWQ <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed 64-bit integers in <i>xmm1</i> .
66 0f 38 35 /r PMOVZXDQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 16-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 31 /r VPMOVZXBW <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ <i>xmm1</i> , <i>xmm2/m16</i>	RM	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of <i>xmm2/m16</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 32-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.128.66.0F38.WIG 35 /r VPMOVZXDQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 2 packed 64-bit integers in <i>xmm1</i> .
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW <i>ymm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX2	Zero extend 16 packed 8-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 16 packed 16-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 31 /r VPMOVZXBW <i>ymm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 8 packed 32-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ <i>ymm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of <i>xmm2/m32</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD <i>ymm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX2	Zero extend 8 packed 16-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 8 packed 32-bit integers in <i>ymm1</i> .

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of <i>xmm2/m64</i> to 4 packed 64-bit integers in <i>ymm1</i> .
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ <i>ymm1, xmm2/m128</i>	RM	V/V	AVX2	Zero extend 4 packed 32-bit integers in the low 16 bytes of <i>xmm2/m128</i> to 4 packed 64-bit integers in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Zero-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination register is YMM Register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMOVZXBW

```
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);
```

PMOVZXBQ

```
DEST[31:0] ← ZeroExtend(SRC[7:0]);
DEST[63:32] ← ZeroExtend(SRC[15:8]);
DEST[95:64] ← ZeroExtend(SRC[23:16]);
DEST[127:96] ← ZeroExtend(SRC[31:24]);
```

PMOVZXQB

```
DEST[63:0] ← ZeroExtend(SRC[7:0]);
DEST[127:64] ← ZeroExtend(SRC[15:8]);
```

PMOVZXWD

```
DEST[31:0] ← ZeroExtend(SRC[15:0]);
DEST[63:32] ← ZeroExtend(SRC[31:16]);
DEST[95:64] ← ZeroExtend(SRC[47:32]);
DEST[127:96] ← ZeroExtend(SRC[63:48]);
```

PMOVZXWQ

DEST[63:0] \leftarrow ZeroExtend(SRC[15:0]);
 DEST[127:64] \leftarrow ZeroExtend(SRC[31:16]);

PMOVZXDQ

DEST[63:0] \leftarrow ZeroExtend(SRC[31:0]);
 DEST[127:64] \leftarrow ZeroExtend(SRC[63:32]);

VPMOVZXBW (VEX.128 encoded version)

Packed_Zero_Extend_BYTE_to_WORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVZXBD (VEX.128 encoded version)

Packed_Zero_Extend_BYTE_to_DWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVZXBQ (VEX.128 encoded version)

Packed_Zero_Extend_BYTE_to_QWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVZXWD (VEX.128 encoded version)

Packed_Zero_Extend_WORD_to_DWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVZXWQ (VEX.128 encoded version)

Packed_Zero_Extend_WORD_to_QWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVZXDQ (VEX.128 encoded version)

Packed_Zero_Extend_DWORD_to_QWORD()
 DEST[VLMAX-1:128] \leftarrow 0

VPMOVZXBW (VEX.256 encoded version)

Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
 Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])

VPMOVZXBD (VEX.256 encoded version)

Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
 Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])

VPMOVZXBQ (VEX.256 encoded version)

Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
 Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])

VPMOVZXWD (VEX.256 encoded version)

Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
 Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])

VPMOVZXWQ (VEX.256 encoded version)

Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
 Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])

VPMOVZXDQ (VEX.256 encoded version)

Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])

Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMOVZXBW: __m128i __mm_cvtepu8_epi16 (__m128i a);
 VPMOVZXBW: __m256i __mm256_cvtepu8_epi16 (__m128i a);
 (V)PMOVZXBD: __m128i __mm_cvtepu8_epi32 (__m128i a);
 VPMOVZXBD: __m256i __mm256_cvtepu8_epi32 (__m128i a);
 (V)PMOVZXBQ: __m128i __mm_cvtepu8_epi64 (__m128i a);
 VPMOVZXBQ: __m256i __mm256_cvtepu8_epi64 (__m128i a);
 (V)PMOVZXWD: __m128i __mm_cvtepu16_epi32 (__m128i a);
 VPMOVZXWD: __m256i __mm256_cvtepu16_epi32 (__m128i a);
 (V)PMOVZXWQ: __m128i __mm_cvtepu16_epi64 (__m128i a);
 VPMOVZXWQ: __m256i __mm256_cvtepu16_epi64 (__m128i a);
 (V)PMOVZXDQ: __m128i __mm_cvtepu32_epi64 (__m128i a);
 VPMOVZXDQ: __m256i __mm256_cvtepu32_epi64 (__m128i a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PMULDQ — Multiply Packed Signed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Multiply the packed signed dword integers in <i>xmm1</i> and <i>xmm2/m128</i> and store the quadword product in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply packed signed doubleword integers in <i>xmm2</i> by packed signed doubleword integers in <i>xmm3/m128</i> , and store the quadword results in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply packed signed doubleword integers in <i>ymm2</i> by packed signed doubleword integers in <i>ymm3/m256</i> , and store the quadword results in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand. For PMULDQ and VPMULDQ (VEX.128 encoded version), the second source operand is two packed signed doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. The first source operand is two packed signed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed signed quadword integers stored in an XMM register. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

For VPMULDQ (VEX.256 encoded version), the second source operand is four packed signed doubleword integers stored in the first (low), third, fifth and seventh doublewords of an YMM register or a 256-bit memory location. The first source operand is four packed signed doubleword integers stored in the first, third, fifth and seventh doublewords of an XMM register. The destination contains four packed signed quadword integers stored in an YMM register. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation.

When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Operation

PMULDQ (128-bit Legacy SSE version)

DEST[63:0] ← DEST[31:0] * SRC[31:0]

DEST[127:64] ← DEST[95:64] * SRC[95:64]

DEST[VLMAX-1:128] (Unmodified)

VPMULDQ (VEX.128 encoded version)

$DEST[63:0] \leftarrow SRC1[31:0] * SRC2[31:0]$
 $DEST[127:64] \leftarrow SRC1[95:64] * SRC2[95:64]$
 $DEST[VLMAX-1:128] \leftarrow 0$

VPMULDQ (VEX.256 encoded version)

$DEST[63:0] \leftarrow SRC1[31:0] * SRC2[31:0]$
 $DEST[127:64] \leftarrow SRC1[95:64] * SRC2[95:64]$
 $DEST[191:128] \leftarrow SRC1[159:128] * SRC2[159:128]$
 $DEST[255:192] \leftarrow SRC1[223:192] * SRC2[223:192]$

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULDQ: __m128i _mm_mul_epi32(__m128i a, __m128i b);
VPMULDQ: __m256i _mm256_mul_epi32(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PMULHRW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 0B /r ¹ PMULHRW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>mm1</i> .
66 0F 38 0B /r PMULHRW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PMULHRW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Legacy SSE version: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Operation

PMULHRSW (with 64-bit operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
```

PMULHRSW (with 128-bit operand)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX-1:128] ← 0
```

VPMULHRSW (VEX.256 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
```

```

temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[175:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240]) >>14) + 1

```

Intel C/C++ Compiler Intrinsic Equivalents

```

PMULHRSW:      __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW:   __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW:     __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E4 /r ¹ PMULHUW mm1, mm2/m64	RM	V/V	SSE	Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E4 /r PMULHUW xmm1, xmm2/m128	RM	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

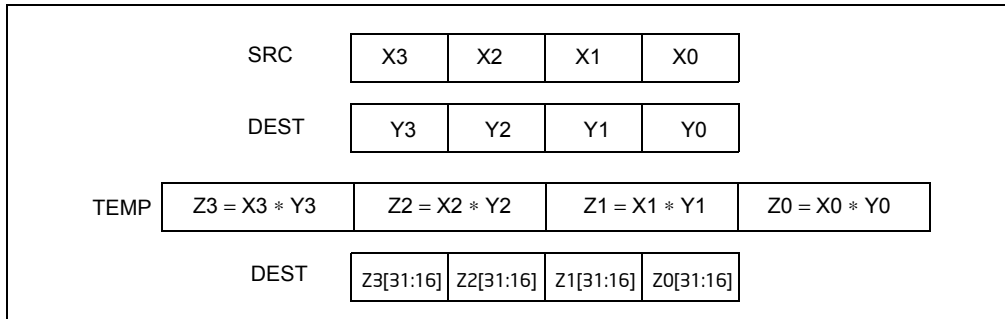


Figure 4-8. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];

```

PMULHUW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];

```

VPMULHUW (VEX.128 encoded version)

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]

```

DEST[63:48] \leftarrow TEMP3[31:16]
 DEST[79:64] \leftarrow TEMP4[31:16]
 DEST[95:80] \leftarrow TEMP5[31:16]
 DEST[111:96] \leftarrow TEMP6[31:16]
 DEST[127:112] \leftarrow TEMP7[31:16]
 DEST[VLMAX-1:128] \leftarrow 0

PMULHUW (VEX.256 encoded version)

TEMP0[31:0] \leftarrow SRC1[15:0] * SRC2[15:0]
 TEMP1[31:0] \leftarrow SRC1[31:16] * SRC2[31:16]
 TEMP2[31:0] \leftarrow SRC1[47:32] * SRC2[47:32]
 TEMP3[31:0] \leftarrow SRC1[63:48] * SRC2[63:48]
 TEMP4[31:0] \leftarrow SRC1[79:64] * SRC2[79:64]
 TEMP5[31:0] \leftarrow SRC1[95:80] * SRC2[95:80]
 TEMP6[31:0] \leftarrow SRC1[111:96] * SRC2[111:96]
 TEMP7[31:0] \leftarrow SRC1[127:112] * SRC2[127:112]
 TEMP8[31:0] \leftarrow SRC1[143:128] * SRC2[143:128]
 TEMP9[31:0] \leftarrow SRC1[159:144] * SRC2[159:144]
 TEMP10[31:0] \leftarrow SRC1[175:160] * SRC2[175:160]
 TEMP11[31:0] \leftarrow SRC1[191:176] * SRC2[191:176]
 TEMP12[31:0] \leftarrow SRC1[207:192] * SRC2[207:192]
 TEMP13[31:0] \leftarrow SRC1[223:208] * SRC2[223:208]
 TEMP14[31:0] \leftarrow SRC1[239:224] * SRC2[239:224]
 TEMP15[31:0] \leftarrow SRC1[255:240] * SRC2[255:240]
 DEST[15:0] \leftarrow TEMP0[31:16]
 DEST[31:16] \leftarrow TEMP1[31:16]
 DEST[47:32] \leftarrow TEMP2[31:16]
 DEST[63:48] \leftarrow TEMP3[31:16]
 DEST[79:64] \leftarrow TEMP4[31:16]
 DEST[95:80] \leftarrow TEMP5[31:16]
 DEST[111:96] \leftarrow TEMP6[31:16]
 DEST[127:112] \leftarrow TEMP7[31:16]
 DEST[143:128] \leftarrow TEMP8[31:16]
 DEST[159:144] \leftarrow TEMP9[31:16]
 DEST[175:160] \leftarrow TEMP10[31:16]
 DEST[191:176] \leftarrow TEMP11[31:16]
 DEST[207:192] \leftarrow TEMP12[31:16]
 DEST[223:208] \leftarrow TEMP13[31:16]
 DEST[239:224] \leftarrow TEMP14[31:16]
 DEST[255:240] \leftarrow TEMP15[31:16]

Intel C/C++ Compiler Intrinsic Equivalent

PMULHUW: `__m64 _mm_mulhi_pu16(__m64 a, __m64 b)`
 (V)PMULHUW: `__m128i _mm_mulhi_epu16 (__m128i a, __m128i b)`
 VPMULHUW: `__m256i _mm256_mulhi_epu16 (__m256i a, __m256i b)`

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E5 /r ¹ PMULHW <i>mm</i> , <i>mm</i> /m64	RM	V/V	MMX	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 0F E5 /r PMULHW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG E5 /r VPMULHW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

n 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Operation

PMULHW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];

```

DEST[31:16] ← TEMP1[31:16];
 DEST[47:32] ← TEMP2[31:16];
 DEST[63:48] ← TEMP3[31:16];

PMULHW (with 128-bit operands)

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
 TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
 TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
 TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
 TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
 TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
 TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
 TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
 DEST[15:0] ← TEMP0[31:16];
 DEST[31:16] ← TEMP1[31:16];
 DEST[47:32] ← TEMP2[31:16];
 DEST[63:48] ← TEMP3[31:16];
 DEST[79:64] ← TEMP4[31:16];
 DEST[95:80] ← TEMP5[31:16];
 DEST[111:96] ← TEMP6[31:16];
 DEST[127:112] ← TEMP7[31:16];

VPMULHW (VEX.128 encoded version)

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
 TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
 TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
 TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
 TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
 TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
 TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
 TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
 DEST[15:0] ← TEMP0[31:16]
 DEST[31:16] ← TEMP1[31:16]
 DEST[47:32] ← TEMP2[31:16]
 DEST[63:48] ← TEMP3[31:16]
 DEST[79:64] ← TEMP4[31:16]
 DEST[95:80] ← TEMP5[31:16]
 DEST[111:96] ← TEMP6[31:16]
 DEST[127:112] ← TEMP7[31:16]
 DEST[VLMAX-1:128] ← 0

PMULHW (VEX.256 encoded version)

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
 TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
 TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
 TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
 TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
 TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
 TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
 TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
 TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
 TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
 TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
 TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
 TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]

TEMP13[31:0] \leftarrow SRC1[223:208] * SRC2[223:208]
 TEMP14[31:0] \leftarrow SRC1[239:224] * SRC2[239:224]
 TEMP15[31:0] \leftarrow SRC1[255:240] * SRC2[255:240]
 DEST[15:0] \leftarrow TEMP0[31:16]
 DEST[31:16] \leftarrow TEMP1[31:16]
 DEST[47:32] \leftarrow TEMP2[31:16]
 DEST[63:48] \leftarrow TEMP3[31:16]
 DEST[79:64] \leftarrow TEMP4[31:16]
 DEST[95:80] \leftarrow TEMP5[31:16]
 DEST[111:96] \leftarrow TEMP6[31:16]
 DEST[127:112] \leftarrow TEMP7[31:16]
 DEST[143:128] \leftarrow TEMP8[31:16]
 DEST[159:144] \leftarrow TEMP9[31:16]
 DEST[175:160] \leftarrow TEMP10[31:16]
 DEST[191:176] \leftarrow TEMP11[31:16]
 DEST[207:192] \leftarrow TEMP12[31:16]
 DEST[223:208] \leftarrow TEMP13[31:16]
 DEST[239:224] \leftarrow TEMP14[31:16]
 DEST[255:240] \leftarrow TEMP15[31:16]

Intel C/C++ Compiler Intrinsic Equivalent

PMULHW: `__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)`
 (V)PMULHW: `__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)`
 VPMULHW: `__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULLD — Multiply Packed Signed Dword Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Multiply the packed dword signed integers in <i>xmm1</i> and <i>xmm2/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 40 /r VPMULLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed dword signed integers in <i>xmm2</i> and <i>xmm3/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 40 /r VPMULLD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed dword signed integers in <i>ymm2</i> and <i>ymm3/m256</i> and store the low 32 bits of each product in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs four signed multiplications from four pairs of signed dword integers and stores the lower 32 bits of the four 64-bit products in the destination operand (first operand). Each dword element in the destination operand is multiplied with the corresponding dword element of the source operand (second operand) to obtain a 64-bit intermediate product.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
Temp0[63:0] ← DEST[31:0] * SRC[31:0];
Temp1[63:0] ← DEST[63:32] * SRC[63:32];
Temp2[63:0] ← DEST[95:64] * SRC[95:64];
Temp3[63:0] ← DEST[127:96] * SRC[127:96];
DEST[31:0] ← Temp0[31:0];
DEST[63:32] ← Temp1[31:0];
DEST[95:64] ← Temp2[31:0];
DEST[127:96] ← Temp3[31:0];
```

VPMULLD (VEX.128 encoded version)

```
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
DEST[31:0] ← Temp0[31:0]
```

$\text{DEST}[63:32] \leftarrow \text{Temp1}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{Temp2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{Temp3}[31:0]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPMULLD (VEX.256 encoded version)

$\text{Temp0}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$
 $\text{Temp1}[63:0] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$
 $\text{Temp2}[63:0] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$
 $\text{Temp3}[63:0] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$
 $\text{Temp4}[63:0] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$
 $\text{Temp5}[63:0] \leftarrow \text{SRC1}[191:160] * \text{SRC2}[191:160]$
 $\text{Temp6}[63:0] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$
 $\text{Temp7}[63:0] \leftarrow \text{SRC1}[255:224] * \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULLUD: `__m128i _mm_mullo_epi32(__m128i a, __m128i b);`
 VPMULLD: `__m256i _mm256_mullo_epi32(__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D5 /r ¹ PMULLW <i>mm</i> , <i>mm</i> /m64	RM	V/V	MMX	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the low 16 bits of the results in <i>mm1</i> .
66 0F D5 /r PMULLW <i>xmm1</i> , <i>xmm2</i> /m128	RM	V/V	SSE2	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the low 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i> /m128	RVM	V/V	AVX	Multiply the packed dword signed integers in <i>xmm2</i> and <i>xmm3/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG D5 /r VPMULLW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3</i> /m256	RVM	V/V	AVX2	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the low 16 bits of the results in <i>ymm1</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

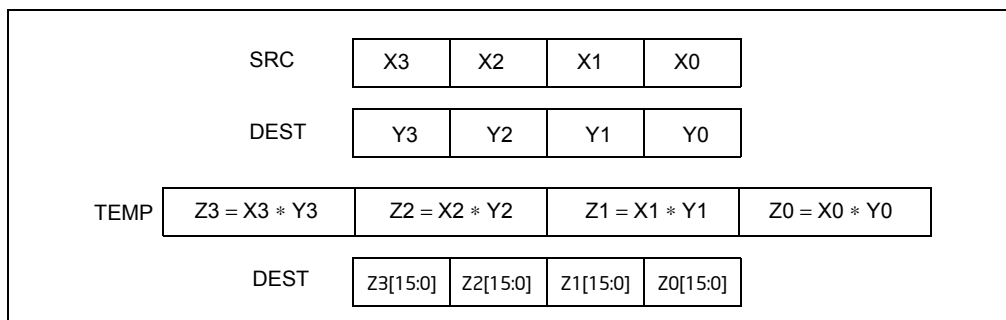


Figure 4-9. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];

```

PMULLW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
DEST[79:64] ← TEMP4[15:0];
DEST[95:80] ← TEMP5[15:0];
DEST[111:96] ← TEMP6[15:0];
DEST[127:112] ← TEMP7[15:0];

```

VPMULLW (VEX.128 encoded version)

```

Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]

```

$\text{DEST}[63:48] \leftarrow \text{Temp3}[15:0]$
 $\text{DEST}[79:64] \leftarrow \text{Temp4}[15:0]$
 $\text{DEST}[95:80] \leftarrow \text{Temp5}[15:0]$
 $\text{DEST}[111:96] \leftarrow \text{Temp6}[15:0]$
 $\text{DEST}[127:112] \leftarrow \text{Temp7}[15:0]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPMULLD (VEX.256 encoded version)

$\text{Temp0}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$
 $\text{Temp1}[63:0] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$
 $\text{Temp2}[63:0] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$
 $\text{Temp3}[63:0] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$
 $\text{Temp4}[63:0] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$
 $\text{Temp5}[63:0] \leftarrow \text{SRC1}[191:160] * \text{SRC2}[191:160]$
 $\text{Temp6}[63:0] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$
 $\text{Temp7}[63:0] \leftarrow \text{SRC1}[255:224] * \text{SRC2}[255:224]$

$\text{DEST}[31:0] \leftarrow \text{Temp0}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{Temp1}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{Temp2}[31:0]$
 $\text{DEST}[127:96] \leftarrow \text{Temp3}[31:0]$
 $\text{DEST}[159:128] \leftarrow \text{Temp4}[31:0]$
 $\text{DEST}[191:160] \leftarrow \text{Temp5}[31:0]$
 $\text{DEST}[223:192] \leftarrow \text{Temp6}[31:0]$
 $\text{DEST}[255:224] \leftarrow \text{Temp7}[31:0]$

Intel C/C++ Compiler Intrinsic Equivalent

PMULLW: `__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)`
 (V)PMULLW: `__m128i _mm_mullo_epi16 (__m128i a, __m128i b)`
 VPMULLW: `__m256i _mm256_mullo_epi16 (__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF F4 /r ¹ PMULUDQ mm1, mm2/m64	RM	V/V	SSE2	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.
66 OF F4 /r PMULUDQ xmm1, xmm2/m128	RM	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.OF.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.OF.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register. The result is an unsigned quadword integer stored in the destination MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first

source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMULUDQ (with 64-Bit operands)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0];$$

PMULUDQ (with 128-Bit operands)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0];$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[95:64] * \text{SRC}[95:64];$$

VPMULUDQ (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$

VPMULUDQ (VEX.256 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$$

$$\text{DEST}[191:128] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$$

$$\text{DEST}[255:192] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$$

Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ: `__m64 _mm_mul_su32 (__m64 a, __m64 b)`

(V)PMULUDQ: `__m128i _mm_mul_epu32 (__m128i a, __m128i b)`

VPMULUDQ: `__m256i _mm256_mul_epu32 (__m256i a, __m256i b);`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

POP—Pop a Value from the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP <i>r/m16</i>	M	Valid	Valid	Pop top of stack into <i>m16</i> ; increment stack pointer.
8F /0	POP <i>r/m32</i>	M	N.E.	Valid	Pop top of stack into <i>m32</i> ; increment stack pointer.
8F /0	POP <i>r/m64</i>	M	Valid	N.E.	Pop top of stack into <i>m64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
58+ <i>rw</i>	POP <i>r16</i>	O	Valid	Valid	Pop top of stack into <i>r16</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r32</i>	O	N.E.	Valid	Pop top of stack into <i>r32</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r64</i>	O	Valid	N.E.	Pop top of stack into <i>r64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	NP	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	NP	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	NP	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	NP	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	NP	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	NP	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	NP	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	NP	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	NP	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
O	opcode + rd (w)	NA	NA	NA
NP	NA	NA	NA	NA

Description

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

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

The address size is used only when writing to a destination operand in memory.

- **Operand size.** The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- **Stack-address size.** Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

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

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

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

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

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

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt¹. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST ← SS:ESP; (* Copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST ← SS:ESP; (* Copy a word *)
```

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before POP ESP executes:

```
POP SS
POP SS
POP ESP
```

```

        ESP ← ESP + 2;
    FI;
ELSE IF StackAddrSize = 64
    THEN
        IF OperandSize = 64
            THEN
                DEST ← SS:RSP; (* Copy quadword *)
                RSP ← RSP + 8;
            ELSE (* OperandSize = 16*)
                DEST ← SS:RSP; (* Copy a word *)
                RSP ← RSP + 2;
            FI;
        FI;
ELSE StackAddrSize = 16
    THEN
        IF OperandSize = 16
            THEN
                DEST ← SS:SP; (* Copy a word *)
                SP ← SP + 2;
            ELSE (* OperandSize = 32 *)
                DEST ← SS:SP; (* Copy a doubleword *)
                SP ← SP + 4;
            FI;
        FI;
FI;

```

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

64-BIT_MODE

```

IF 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 FS, or GS is loaded with a NULL selector;
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;

```

PREOTECTED MODE OR COMPATIBILITY MODE;

```

IF SS is loaded;

```

```

THEN
    IF segment selector is NULL
        THEN #GP(0);
    FI;
    IF segment selector index is outside descriptor table limits
        or segment selector's RPL  $\neq$  CPL
        or segment is not a writable data segment
        or DPL  $\neq$  CPL
        THEN #GP(selector);
    FI;
    IF segment not marked present
        THEN #SS(selector);
    ELSE
        SS  $\leftarrow$  segment selector;
        SS  $\leftarrow$  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);
    FI;
    IF segment not marked present
        THEN #NP(selector);
    ELSE
        SegmentRegister  $\leftarrow$  segment selector;
        SegmentRegister  $\leftarrow$  segment descriptor;
    FI;
FI;

IF DS, ES, FS, or GS is loaded with a NULL selector
THEN
    SegmentRegister  $\leftarrow$  segment selector;
    SegmentRegister  $\leftarrow$  segment descriptor;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If attempt is made to load SS register with NULL segment selector.</p> <p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p>

	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.

POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
61	POPA	NP	Invalid	Valid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	NP	Invalid	Valid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

IF OperandSize = 32 (* Instruction = POPAD *)

THEN

EDI ← Pop();

ESI ← Pop();

EBP ← Pop();

Increment ESP by 4; (* Skip next 4 bytes of stack *)

EBX ← Pop();

EDX ← Pop();

ECX ← Pop();

EAX ← Pop();

ELSE (* OperandSize = 16, instruction = POPA *)

DI ← Pop();

SI ← Pop();

BP ← Pop();

Increment ESP by 2; (* Skip next 2 bytes of stack *)

BX ← Pop();

DX ← Pop();

CX ← Pop();

AX ← Pop();

FI;

FI;

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.
-----	--------------------

POPCNT — Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 0F B8 /r	POPCNT <i>r16, r/m16</i>	RM	Valid	Valid	POPCNT on <i>r/m16</i>
F3 0F B8 /r	POPCNT <i>r32, r/m32</i>	RM	Valid	Valid	POPCNT on <i>r/m32</i>
F3 REX.W 0F B8 /r	POPCNT <i>r64, r/m64</i>	RM	Valid	N.E.	POPCNT on <i>r/m64</i>

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{
    IF (SRC[ i] = 1) // i'th bit
        THEN Count++;
}
DEST ← Count;
```

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

Intel C/C++ Compiler Intrinsic Equivalent

```
POPCNT:    int __mm_popcnt_u32(unsigned int a);
POPCNT:    int64_t __mm_popcnt_u64(unsigned __int64 a);
```

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
 #SS(0) If a memory operand effective address is outside the SS segment limit.
 #PF (fault-code) For a page fault.
 #AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
 #UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
 If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
 #SS(0) If a memory operand effective address is outside the SS segment limit.
 #UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
 If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used.

POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9D	POPF	NP	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	NP	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPFQ	NP	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See the Table 4-12 and key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF¹, VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF¹, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

In 64-bit mode, use REX.W to pop the top of stack to RFLAGS. The mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack, loads the lower 32 bits into RFLAGS, and zero extends the upper bits of RFLAGS.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS registers.

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

Table 4-12. Effect of POPF/POPFD on the EFLAGS Register

Mode	Operand Size	CPL	IOPL	Flags																	Notes
				21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0	
				ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF	
Real-Address Mode (CR0.PE = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
Protected, Compatibility, and 64-Bit Modes (CR0.PE = 1, EFLAGS.VM = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	16	1-3	<CPL	N	N	N	N	N	0	S	N	S	S	N	S	S	S	S	S	S	
	16	1-3	≥CPL	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32, 64	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32, 64	1-3	<CPL	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S	
	32, 64	1-3	≥CPL	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
Virtual-8086 (CR0.PE = 1, EFLAGS.VM = 1, CR4.VME = 0)	16	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
VME (CR0.PE = 1, EFLAGS.VM = 1, CR4.VME = 1)	16	3	0-2	N/X	N/X	SV/X	N/X	N/X	0/X	S/X	N/X	S/X	S/X	N/X	S/X	S/X	S/X	S/X	S/X	S/X	2
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	

NOTES:

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack

Key	
S	Updated from stack
SV	Updated from IF (bit 9) in FLAGS value on stack
N	No change in value
X	No EFLAGS update
0	Value is cleared

Operation

IF VM = 0 (* Not in Virtual-8086 Mode *)

THEN IF CPL = 0

THEN

IF OperandSize = 32;

THEN

EFLAGS ← Pop(); (* 32-bit pop *)

(* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)

ELSE IF (OperandSize = 64)

RFLAGS = Pop(); (* 64-bit pop *)

(* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)

ELSE (* OperandSize = 16 *)

```

        EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
        (* All non-reserved flags can be modified. *)
    FI;
ELSE (* CPL > 0 *)
    IF OperandSize = 32
        THEN
            IF CPL > IOPL
                THEN
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                    IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                ELSE
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                    IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
            FI;
        ELSE IF (OperandSize = 64)
            IF CPL > IOPL
                THEN
                    RFLAGS ← Pop(); (* 64-bit pop *)
                    (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                    IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                ELSE
                    RFLAGS ← Pop(); (* 64-bit pop *)
                    (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                    IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
            FI;
        ELSE (* OperandSize = 16 *)
            EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
            (* All non-reserved bits except IOPL can be modified; IOPL and all
            reserved bits are unaffected. *)
        FI;
    FI;
ELSE IF CR4.VME = 1 (* In Virtual-8086 Mode with VME Enabled *)
    IF IOPL = 3
        THEN IF OperandSize = 32
            THEN
                EFLAGS ← Pop();
                (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
            ELSE
                EFLAGS[15:0] ← Pop(); FI;
                (* All non-reserved bits except IOPL can be modified;
                IOPL and all reserved bits are unaffected. *)
        FI;
    ELSE (* IOPL < 3 *)
        IF (OperandSize = 32)
            THEN
                #GP(0); (* Trap to virtual-8086 monitor. *)
            ELSE (* OperandSize = 16 *)
                tempFLAGS ← Pop();
                IF EFLAGS.VIP = 1 AND tempFLAGS[9] = 1
                    THEN #GP(0);
                ELSE

```

```

        EFLAGS.VIF ← tempFLAGS[9];
        EFLAGS[15:0] ← tempFLAGS;
        (* All non-reserved bits except IOPL and IF can be modified;
        IOPL, IF, and all reserved bits are unaffected. *)
    FI;
FI;
FI;
ELSE (* In Virtual-8086 Mode *)
    IF IOPL = 3
        THEN IF OperandSize = 32
            THEN
                EFLAGS ← Pop();
                (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
            ELSE
                EFLAGS[15:0] ← Pop(); FI;
                (* All non-reserved bits except IOPL can be modified;
                IOPL and all reserved bits are unaffected. *)
        ELSE (* IOPL < 3 *)
            #GP(0); (* Trap to virtual-8086 monitor. *)
    FI;
FI;
FI;

```

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3. If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EB /r ¹ POR <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Bitwise OR of <i>mm/m64</i> and <i>mm</i> .
66 0F EB /r POR <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG EB /r VPOR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Bitwise OR of <i>xmm2/m128</i> and <i>xmm3</i> .
VEX.NDS.256.66.0F.WIG EB /r VPOR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Bitwise OR of <i>ymm2/m256</i> and <i>ymm3</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

POR (128-bit Legacy SSE version)

DEST ← DEST OR SRC

DEST[VLMAX-1:128] (Unmodified)

VPOR (VEX.128 encoded version)

DEST ← SRC1 OR SRC2

DEST[VLMAX-1:128] ← 0

VPOR (VEX.256 encoded version)

DEST ← SRC1 OR SRC2

Intel C/C++ Compiler Intrinsic Equivalent

POR: __m64 _mm_or_si64(__m64 m1, __m64 m2)

(V)POR: __m128i _mm_or_si128(__m128i m1, __m128i m2)

VPOR: __m256i _mm256_or_si256 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PREFETCHh—Prefetch Data Into Caches

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 18 /1	PREFETCHT0 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T0 hint.
OF 18 /2	PREFETCHT1 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T1 hint.
OF 18 /3	PREFETCHT2 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T2 hint.
OF 18 /0	PREFETCHNTA <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using NTA hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
 - Pentium III processor—1st- or 2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
 - Pentium III processor—2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
 - Pentium III processor—2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
 - Pentium III processor—1st-level cache
 - Pentium 4 and Intel Xeon processors—2nd-level cache

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also

unordered with respect to CLFLUSH instructions, other PREFETCH h instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

FETCH (m8);

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch(char *p, int i)
```

The argument “ p ” gives the address of the byte (and corresponding cache line) to be prefetched. The value “ i ” gives a constant (`_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, or `_MM_HINT_NTA`) that specifies the type of prefetch operation to be performed.

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 0D /1 PREFETCHW m8	A	V/V	PRFCHW	Move data from m8 closer to the processor in anticipation of a write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates all other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates any other cached copy in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH instructions, other PREFETCHW instructions, or any other general instruction.

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

```
void _m_prefetchw( void * );
```

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.

PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 0D /2 PREFETCHWT1 m8	M	V/V	PREFETCHWT1	Move data from m8 closer to the processor using T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

- T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHH instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHH instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHH instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHH instruction is also unordered with respect to CLFLUSH instructions, other PREFETCHH instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

Prefetch (m8, Level = 1, EXCLUSIVE=1);

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch( char const *, int hint= _MM_HINT_ET1);
```

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F6 /r ¹ PSADBW mm1, mm2/m64	RM	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2/m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW xmm1, xmm2/m128	RM	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2/m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3/m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3/m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-10 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

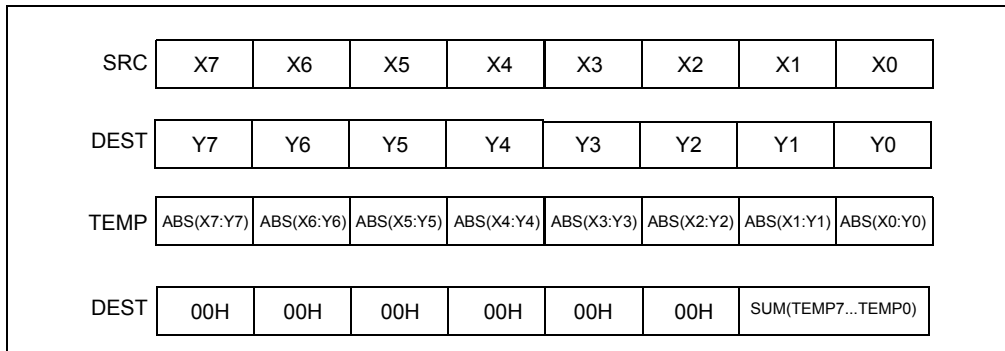


Figure 4-10. PSADBW Instruction Operation Using 64-bit Operands

Operation

PSADBW (when using 64-bit operands)

```

TEMPO ← ABS(DEST[7:0] – SRC[7:0]);
(* Repeat operation for bytes 2 through 6 *)
TEMP7 ← ABS(DEST[63:56] – SRC[63:56]);
DEST[15:0] ← SUM(TEMPO:TEMP7);
DEST[63:16] ← 000000000000H;

```

PSADBW (when using 128-bit operands)

```

TEMPO ← ABS(DEST[7:0] – SRC[7:0]);
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] – SRC[127:120]);
DEST[15:0] ← SUM(TEMPO:TEMP7);
DEST[63:16] ← 000000000000H;
DEST[79:64] ← SUM(TEMP8:TEMP15);
DEST[127:80] ← 000000000000H;
DEST[VLMAX-1:128] (Unmodified)

```

VPSADBW (VEX.128 encoded version)

```

TEMPO ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ← SUM(TEMPO:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 000000000000
DEST[VLMAX-1:128] ← 0

```

VPSADBW (VEX.256 encoded version)

```

TEMPO ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] ← SUM(TEMPO:TEMP7)

```


DEST[63:16] \leftarrow 000000000000H
 DEST[79:64] \leftarrow SUM(TEMP8:TEMP15)
 DEST[127:80] \leftarrow 000000000000H
 DEST[143:128] \leftarrow SUM(TEMP16:TEMP23)
 DEST[191:144] \leftarrow 000000000000H
 DEST[207:192] \leftarrow SUM(TEMP24:TEMP31)
 DEST[223:208] \leftarrow 000000000000H

Intel C/C++ Compiler Intrinsic Equivalent

PSADBW: `__m64 _mm_sad_pu8(__m64 a, __m64 b)`
 (V)PSADBW: `__m128i _mm_sad_epu8(__m128i a, __m128i b)`
 VPSADBW: `__m256i _mm256_sad_epu8(__m256i a, __m256i b)`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 00 /r ¹ PSHUFB <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Shuffle bytes in <i>mm1</i> according to contents of <i>mm2/m64</i> .
66 0F 38 00 /r PSHUFB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Shuffle bytes in <i>xmm1</i> according to contents of <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> .

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Legacy SSE version: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask. The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PSHUFB (with 64 bit operands)

TEMP ← DEST

for i = 0 to 7 {

```

if (SRC[(i * 8)+7] = 1 ) then
    DEST[(i*8)+7...(i*8)+0] ← 0;
else
    index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
    DEST[(i*8)+7...(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
endif;
}

```

PSHUFB (with 128 bit operands)

TEMP ← DEST

```

for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX-1:128] ← 0

```

VPSHUFB (VEX.128 encoded version)

```

for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX-1:128] ← 0

```

VPSHUFB (VEX.256 encoded version)

```

for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7...(i*8)+0] ← 0;
    else
        index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7...(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
    endif
}

```

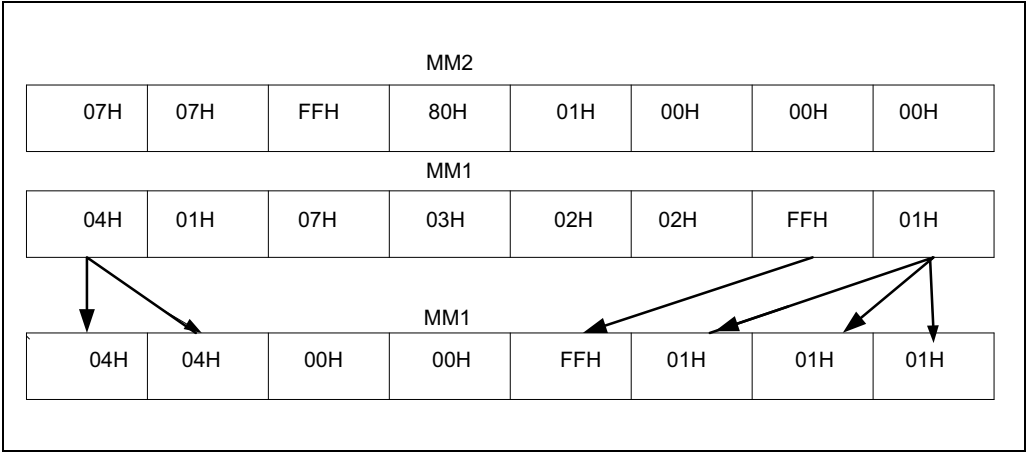


Figure 4-11. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFB:      __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB:   __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:     __m256i _mm256_shuffle_epi8(__m256i a, __m256i b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 1.

PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.66.0F.WIG 70 /r ib VPSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 70 /r ib VPSHUFD <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Shuffle the doublewords in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-12 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-12) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

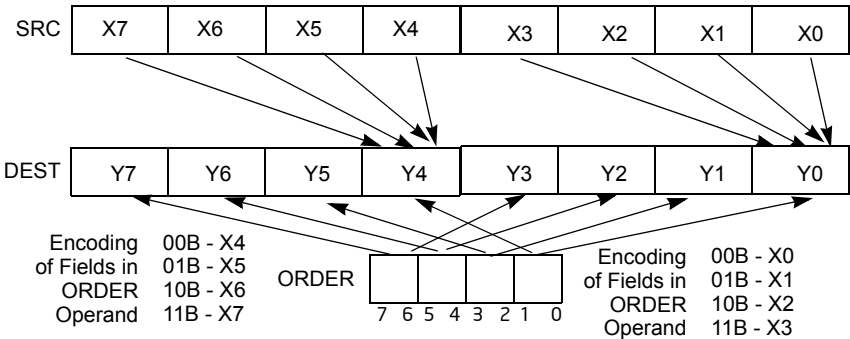


Figure 4-12. 256-bit VPSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

Legacy SSE instructions: In 64-bit mode using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] (Unmodified)
```

VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] ← 0
```

VPSHUFD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFD: `__m128i _mm_shuffle_epi32(__m128i a, int n)`

VPSHUFD: `__m256i _mm256_shuffle_epi32(__m256i a, const int n)`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-12. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

Note: In VEX encoded versions VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFHW (128-bit Legacy SSE version)

```

DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] (Unmodified)

```

VPSHUFHW (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]$
 $\text{DEST}[79:64] \leftarrow (\text{SRC1} \gg (\text{imm}[1:0] * 16))[79:64]$
 $\text{DEST}[95:80] \leftarrow (\text{SRC1} \gg (\text{imm}[3:2] * 16))[79:64]$
 $\text{DEST}[111:96] \leftarrow (\text{SRC1} \gg (\text{imm}[5:4] * 16))[79:64]$
 $\text{DEST}[127:112] \leftarrow (\text{SRC1} \gg (\text{imm}[7:6] * 16))[79:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPSHUFHW (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]$
 $\text{DEST}[79:64] \leftarrow (\text{SRC1} \gg (\text{imm}[1:0] * 16))[79:64]$
 $\text{DEST}[95:80] \leftarrow (\text{SRC1} \gg (\text{imm}[3:2] * 16))[79:64]$
 $\text{DEST}[111:96] \leftarrow (\text{SRC1} \gg (\text{imm}[5:4] * 16))[79:64]$
 $\text{DEST}[127:112] \leftarrow (\text{SRC1} \gg (\text{imm}[7:6] * 16))[79:64]$
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128]$
 $\text{DEST}[207:192] \leftarrow (\text{SRC1} \gg (\text{imm}[1:0] * 16))[207:192]$
 $\text{DEST}[223:208] \leftarrow (\text{SRC1} \gg (\text{imm}[3:2] * 16))[207:192]$
 $\text{DEST}[239:224] \leftarrow (\text{SRC1} \gg (\text{imm}[5:4] * 16))[207:192]$
 $\text{DEST}[255:240] \leftarrow (\text{SRC1} \gg (\text{imm}[7:6] * 16))[207:192]$

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFHW: `__m128i _mm_shufflehi_epi16(__m128i a, int n)`
VPSHUFHW: `__m256i _mm256_shufflehi_epi16(__m256i a, const int n)`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-12. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise instructions will #UD.

Operation

PSHUFLW (128-bit Legacy SSE version)

```

DEST[15:0] ← (SRC >> (imm[1:0] * 16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)

```

VPSHUFLW (VEX.128 encoded version)

$\text{DEST}[15:0] \leftarrow (\text{SRC1} \gg (\text{imm}[1:0] * 16))[15:0]$
 $\text{DEST}[31:16] \leftarrow (\text{SRC1} \gg (\text{imm}[3:2] * 16))[15:0]$
 $\text{DEST}[47:32] \leftarrow (\text{SRC1} \gg (\text{imm}[5:4] * 16))[15:0]$
 $\text{DEST}[63:48] \leftarrow (\text{SRC1} \gg (\text{imm}[7:6] * 16))[15:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPSHUFLW (VEX.256 encoded version)

$\text{DEST}[15:0] \leftarrow (\text{SRC1} \gg (\text{imm}[1:0] * 16))[15:0]$
 $\text{DEST}[31:16] \leftarrow (\text{SRC1} \gg (\text{imm}[3:2] * 16))[15:0]$
 $\text{DEST}[47:32] \leftarrow (\text{SRC1} \gg (\text{imm}[5:4] * 16))[15:0]$
 $\text{DEST}[63:48] \leftarrow (\text{SRC1} \gg (\text{imm}[7:6] * 16))[15:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$
 $\text{DEST}[143:128] \leftarrow (\text{SRC1} \gg (\text{imm}[1:0] * 16))[143:128]$
 $\text{DEST}[159:144] \leftarrow (\text{SRC1} \gg (\text{imm}[3:2] * 16))[143:128]$
 $\text{DEST}[175:160] \leftarrow (\text{SRC1} \gg (\text{imm}[5:4] * 16))[143:128]$
 $\text{DEST}[191:176] \leftarrow (\text{SRC1} \gg (\text{imm}[7:6] * 16))[143:128]$
 $\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192]$

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFLW: `__m128i _mm_shufflelo_epi16(__m128i a, int n)`
VPSHUFLW: `__m256i _mm256_shufflelo_epi16(__m256i a, const int n)`

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
 If VEX.vvvv ≠ 1111B.

PSHUFW—Shuffle Packed Words

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 70 /r ib PSHUFW <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	RMI	Valid	Valid	Shuffle the words in <i>mm2/m64</i> based on the encoding in <i>imm8</i> and store the result in <i>mm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-12. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFW:    __m64 _mm_shuffle_pi16(__m64 a, int n)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Table 22-7, “Exception Conditions for SIMD/MMX Instructions with Memory Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 08 /r ¹ PSIGNB <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m64</i> .
66 0F 38 08 /r PSIGNB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
0F 38 09 /r ¹ PSIGNW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m128</i> .
66 0F 38 09 /r PSIGNW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
0F 38 0A /r ¹ PSIGND <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m128</i> .
66 0F 38 0A /r PSIGND <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed word integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 08 /r VPSIGNB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Negate packed byte integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
VEX.NDS.256.66.0F38.WIG 09 /r VPSIGNW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Negate packed 16-bit integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
VEX.NDS.256.66.0F38.WIG 0A /r VPSIGND <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Negate packed doubleword integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Operation

PSIGNB (with 64 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes
```

```
IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]
```

PSIGNB (with 128 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]
```

VPSIGNB (VEX.128 encoded version)

$$\text{DEST}[127:0] \leftarrow \text{BYTE_SIGN}(\text{SRC1}, \text{SRC2})$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VPSIGNB (VEX.256 encoded version)

$$\text{DEST}[255:0] \leftarrow \text{BYTE_SIGN_256b}(\text{SRC1}, \text{SRC2})$$
PSIGNW (with 64 bit operands)

$$\text{IF } (\text{SRC}[15:0] < 0)$$

$$\text{DEST}[15:0] \leftarrow \text{Neg}(\text{DEST}[15:0])$$

$$\text{ELSEIF } (\text{SRC}[15:0] = 0)$$

$$\text{DEST}[15:0] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[15:0] > 0)$$

$$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0]$$

Repeat operation for 2nd through 3rd words

$$\text{IF } (\text{SRC}[63:48] < 0)$$

$$\text{DEST}[63:48] \leftarrow \text{Neg}(\text{DEST}[63:48])$$

$$\text{ELSEIF } (\text{SRC}[63:48] = 0)$$

$$\text{DEST}[63:48] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[63:48] > 0)$$

$$\text{DEST}[63:48] \leftarrow \text{DEST}[63:48]$$
PSIGNW (with 128 bit operands)

$$\text{IF } (\text{SRC}[15:0] < 0)$$

$$\text{DEST}[15:0] \leftarrow \text{Neg}(\text{DEST}[15:0])$$

$$\text{ELSEIF } (\text{SRC}[15:0] = 0)$$

$$\text{DEST}[15:0] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[15:0] > 0)$$

$$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0]$$

Repeat operation for 2nd through 7th words

$$\text{IF } (\text{SRC}[127:112] < 0)$$

$$\text{DEST}[127:112] \leftarrow \text{Neg}(\text{DEST}[127:112])$$

$$\text{ELSEIF } (\text{SRC}[127:112] = 0)$$

$$\text{DEST}[127:112] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[127:112] > 0)$$

$$\text{DEST}[127:112] \leftarrow \text{DEST}[127:112]$$
VPSIGNW (VEX.128 encoded version)

$$\text{DEST}[127:0] \leftarrow \text{WORD_SIGN}(\text{SRC1}, \text{SRC2})$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VPSIGNW (VEX.256 encoded version)

$$\text{DEST}[255:0] \leftarrow \text{WORD_SIGN}(\text{SRC1}, \text{SRC2})$$
PSIGND (with 64 bit operands)

$$\text{IF } (\text{SRC}[31:0] < 0)$$

$$\text{DEST}[31:0] \leftarrow \text{Neg}(\text{DEST}[31:0])$$

$$\text{ELSEIF } (\text{SRC}[31:0] = 0)$$

$$\text{DEST}[31:0] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[31:0] > 0)$$

$$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0]$$

$$\text{IF } (\text{SRC}[63:32] < 0)$$

$$\text{DEST}[63:32] \leftarrow \text{Neg}(\text{DEST}[63:32])$$

$$\text{ELSEIF } (\text{SRC}[63:32] = 0)$$

$$\text{DEST}[63:32] \leftarrow 0$$

```
ELSEIF (SRC[63:32] > 0 )
    DEST[63:32] ← DEST[63:32]
```

PSIGND (with 128 bit operands)

```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
Repeat operation for 2nd through 3rd double words
IF (SRC[127:96] < 0 )
    DEST[127:96] ← Neg(DEST[127:96])
ELSEIF (SRC[127:96] = 0 )
    DEST[127:96] ← 0
ELSEIF (SRC[127:96] > 0 )
    DEST[127:96] ← DEST[127:96]
```

VPSIGND (VEX.128 encoded version)

```
DEST[127:0] ← DWORD_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPSIGND (VEX.256 encoded version)

```
DEST[255:0] ← DWORD_SIGN(SRC1, SRC2)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSIGNB:      __m64 _mm_sign_pi8 (__m64 a, __m64 b)
(V)PSIGNB:   __m128i _mm_sign_epi8 (__m128i a, __m128i b)
VPSIGNB:     __m256i _mm256_sign_epi8 (__m256i a, __m256i b)
PSIGNW:      __m64 _mm_sign_pi16 (__m64 a, __m64 b)
(V)PSIGNW:   __m128i _mm_sign_epi16 (__m128i a, __m128i b)
VPSIGNW:     __m256i _mm256_sign_epi16 (__m256i a, __m256i b)
PSIGND:      __m64 _mm_sign_pi32 (__m64 a, __m64 b)
(V)PSIGND:   __m128i _mm_sign_epi32 (__m128i a, __m128i b)
VPSIGND:     __m256i _mm256_sign_epi32 (__m256i a, __m256i b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift <i>xmm1</i> left by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift <i>xmm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift <i>ymm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	imm8	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

Note: VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSLLDQ(128-bit Legacy SSE version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← DEST << (TEMP * 8)

DEST[VLMAX-1:128] (Unmodified)

VPSLLDQ (VEX.128 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← SRC << (TEMP * 8)

DEST[VLMAX-1:128] ← 0

VPSLLDQ (VEX.256 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST[127:0] ← SRC[127:0] << (TEMP * 8)

DEST[255:128] ← SRC[255:128] << (TEMP * 8)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSLLDQ: `__m128i _mm_slli_si128 (__m128i a, int imm)`
VPSLLDQ: `__m256i _mm256_slli_si256 (__m256i a, const int imm)`

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally
#UD If VEX.L = 1.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F1 /r ¹ PSLLW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> left <i>mm/m64</i> while shifting in 0s.
66 0F F1 /r PSLLW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 71 /6 ib PSLLW <i>mm1</i> , <i>imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 71 /6 ib PSLLW <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F2 /r ¹ PSLLD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F2 /r PSLLD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 72 /6 ib ¹ PSLLD <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 72 /6 ib PSLLD <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F3 /r ¹ PSLLQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift quadword in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F3 /r PSLLQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 73 /6 ib ¹ PSLLQ <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift quadword in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 73 /6 ib PSLLQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift quadwords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift quadwords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift doublewords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift doublewords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift quadwords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift quadwords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-13 gives an example of shifting words in a 64-bit operand.

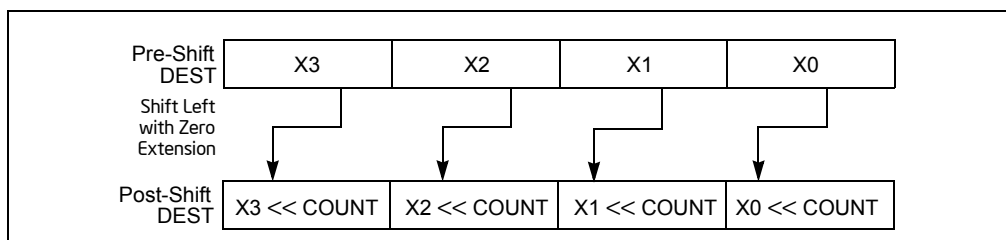


Figure 4-13. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
  FI;
```

PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
  FI;
```

PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST ← ZeroExtend(DEST << COUNT);
  FI;
```

PSLLW (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(DEST[127:112] << COUNT);
  FI;
```

PSLLD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
```

(* Repeat shift operation for 2nd and 3rd doublewords *)
 DEST[127:96] ← ZeroExtend(DEST[127:96] << COUNT);

FI;

PSLLQ (with 128-bit operand)

COUNT ← COUNT_SOURCE[63:0];
 IF (COUNT > 63)
 THEN
 DEST[128:0] ← 00000000000000000000000000000000H;
 ELSE
 DEST[63:0] ← ZeroExtend(DEST[63:0] << COUNT);
 DEST[127:64] ← ZeroExtend(DEST[127:64] << COUNT);
 FI;

PSLLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
 DEST[VLMAX-1:128] (Unmodified)

PSLLW (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
 DEST[VLMAX-1:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
 DEST[VLMAX-1:128] ← 0

VPSLLD (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
 DEST[VLMAX-1:128] ← 0

PSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
 DEST[VLMAX-1:128] (Unmodified)

PSLLD (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
 DEST[VLMAX-1:128] (Unmodified)

VPSLLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
 DEST[VLMAX-1:128] ← 0

VPSLLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
 DEST[VLMAX-1:128] ← 0

PSLLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
 DEST[VLMAX-1:128] (Unmodified)

PSLLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
 DEST[VLMAX-1:128] (Unmodified)

VPSLLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)

DEST[VLMAX-1:128] ← 0

VPSLLW (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)

DEST[VLMAX-1:128] ← 0

PSLLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)

DEST[VLMAX-1:128] (Unmodified)

PSLLW (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)

DEST[VLMAX-1:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)

DEST[VLMAX-1:128] ← 0

VPSLLD (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)

DEST[VLMAX-1:128] ← 0

VPSLLW (ymm, ymm, xmm/m128)

DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)

VPSLLW (ymm, imm8)

DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)

VPSLLD (ymm, ymm, xmm/m128)

DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)

VPSLLD (ymm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)

VPSLLQ (ymm, ymm, xmm/m128)

DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)

VPSLLQ (ymm, imm8)

DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)

Intel C/C++ Compiler Intrinsic EquivalentsPSLLW: `__m64 _mm_slli_pi16 (__m64 m, int count)`PSLLW: `__m64 _mm_sll_pi16 (__m64 m, __m64 count)`(V)PSLLW: `__m128i _mm_slli_pi16 (__m64 m, int count)`(V)PSLLW: `__m128i _mm_slli_pi16 (__m128i m, __m128i count)`VPSLLW: `__m256i _mm256_slli_epi16 (__m256i m, int count)`VPSLLW: `__m256i _mm256_sll_epi16 (__m256i m, __m128i count)`PSLLD: `__m64 _mm_slli_pi32 (__m64 m, int count)`PSLLD: `__m64 _mm_sll_pi32 (__m64 m, __m64 count)`

(V)PSLLD: `__m128i _mm_slli_epi32(__m128i m, int count)`
 (V)PSLLD: `__m128i _mm_sll_epi32(__m128i m, __m128i count)`
 VPSLLD: `__m256i _mm256_slli_epi32 (__m256i m, int count)`
 VPSLLD: `__m256i _mm256_sll_epi32 (__m256i m, __m128i count)`
 PSLLQ: `__m64 _mm_slli_si64(__m64 m, int count)`
 PSLLQ: `__m64 _mm_sll_si64(__m64 m, __m64 count)`
 (V)PSLLQ: `__m128i _mm_slli_epi64(__m128i m, int count)`
 (V)PSLLQ: `__m128i _mm_sll_epi64(__m128i m, __m128i count)`
 VPSLLQ: `__m256i _mm256_slli_epi64 (__m256i m, int count)`
 VPSLLQ: `__m256i _mm256_sll_epi64 (__m256i m, __m128i count)`

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally

#UD If VEX.L = 1.

PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E1 /r ¹ PSRAW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E1 /r PSRAW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 71 /4 ib ¹ PSRAW <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 71 /4 ib PSRAW <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
0F E2 /r ¹ PSRAD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E2 /r PSRAD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift doubleword in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 72 /4 ib ¹ PSRAD <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 72 /4 ib PSRAD <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

Description

Shifts the bits in the individual data elements (words or doublewords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-14 gives an example of shifting words in a 64-bit operand.)

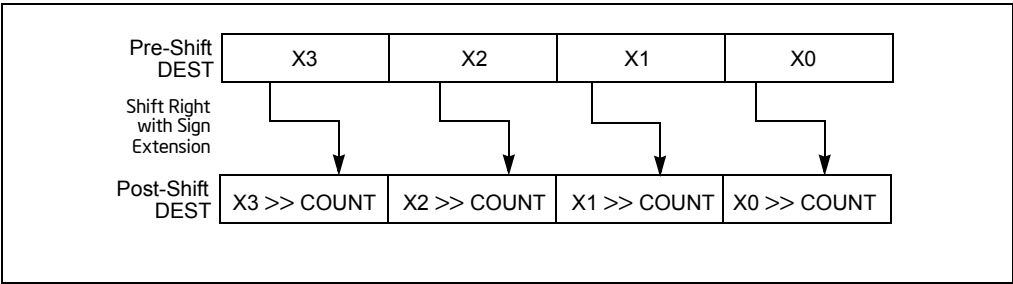


Figure 4-14. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSRAW (with 64-bit operand)

```
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

PSRAD (with 64-bit operand)

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);
```

PSRAW (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(DEST[127:112] >> COUNT);
```

PSRAD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd doublewords *)
DEST[127:96] ← SignExtend(DEST[127:96] >> COUNT);
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0
```

PSRAD (xmm, xmm, xmm/m128)

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)

DEST[VLMAX-1:128] (Unmodified)

PSRAD (xmm, imm8)

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)

DEST[VLMAX-1:128] (Unmodified)

VPSRAD (xmm, xmm, xmm/m128)

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)

DEST[VLMAX-1:128] ← 0

VPSRAD (xmm, imm8)

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)

DEST[VLMAX-1:128] ← 0

VPSRAW (ymm, ymm, xmm/m128)

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)

VPSRAW (ymm, imm8)

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)

VPSRAD (ymm, ymm, xmm/m128)

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)

VPSRAD (ymm, imm8)

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)

Intel C/C++ Compiler Intrinsic Equivalents

PSRAW: __m64 _mm_srai_pi16 (__m64 m, int count)

PSRAW: __m64 _mm_sra_pi16 (__m64 m, __m64 count)

(V)PSRAW: __m128i _mm_srai_epi16(__m128i m, int count)

(V)PSRAW: __m128i _mm_sra_epi16(__m128i m, __m128i count)

VPSRAW: __m256i _mm256_srai_epi16 (__m256i m, int count)

VPSRAW: __m256i _mm256_sra_epi16 (__m256i m, __m128i count)

PSRAD: __m64 _mm_srai_pi32 (__m64 m, int count)

PSRAD: __m64 _mm_sra_pi32 (__m64 m, __m64 count)

(V)PSRAD: __m128i _mm_srai_epi32 (__m128i m, int count)

(V)PSRAD: __m128i _mm_sra_epi32 (__m128i m, __m128i count)

VPSRAD: __m256i _mm256_srai_epi32 (__m256i m, int count)

VPSRAD: __m256i _mm256_sra_epi32 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally
#UD If VEX.L = 1.

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift <i>xmm2</i> right by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift <i>ymm1</i> right by <i>imm8</i> bytes while shifting in 0s.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	imm8	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

Note: VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSRLDQ(128-bit Legacy SSE version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← DEST >> (TEMP * 8)

DEST[VLMAX-1:128] (Unmodified)

VPSRLDQ (VEX.128 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← SRC >> (TEMP * 8)

DEST[VLMAX-1:128] ← 0

VPSRLDQ (VEX.256 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST[127:0] ← SRC[127:0] >> (TEMP * 8)

DEST[255:128] ← SRC[255:128] >> (TEMP * 8)

Intel C/C++ Compiler Intrinsic Equivalents

(V)PSRLDQ: `__m128i _mm_srli_si128 (__m128i a, int imm)`

VPSRLDQ: `__m256i _mm256_srli_si256 (__m256i a, const int imm)`

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.L = 1.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D1 /r ¹ PSRLW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D1 /r PSRLW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 71 /2 ib ¹ PSRLW <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 71 /2 ib PSRLW <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D2 /r ¹ PSRLD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D2 /r PSRLD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 72 /2 ib ¹ PSRLD <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 72 /2 ib PSRLD <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D3 /r ¹ PSRLQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D3 /r PSRLQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 73 /2 ib ¹ PSRLQ <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 73 /2 ib PSRLQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift quadwords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift quadwords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift quadwords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift quadwords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-15 gives an example of shifting words in a 64-bit operand.

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

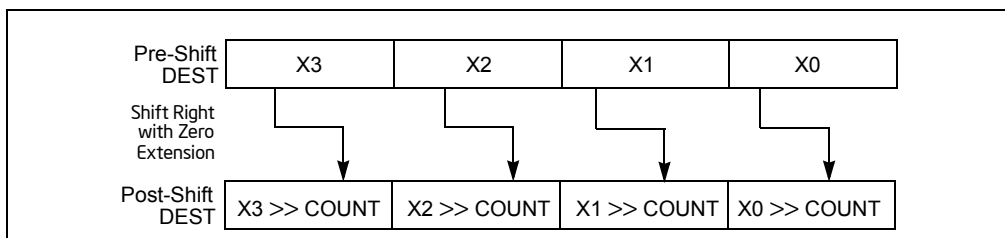


Figure 4-15. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits

are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSRLW (with 64-bit operand)

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
  FI;
```

PSRLD (with 64-bit operand)

```
IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
  FI;
```

PSRLQ (with 64-bit operand)

```
IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST ← ZeroExtend(DEST >> COUNT);
  FI;
```

PSRLW (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(DEST[127:112] >> COUNT);
  FI;
```

PSRLD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H
```

```

ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd doublewords *)
    DEST[127:96] ← ZeroExtend(DEST[127:96] >> COUNT);
FI;

```

PSRLQ (with 128-bit operand)

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H
    ELSE
        DEST[63:0] ← ZeroExtend(DEST[63:0] >> COUNT);
        DEST[127:64] ← ZeroExtend(DEST[127:64] >> COUNT);
FI;

```

PSRLW (xmm, xmm, xmm/m128)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

PSRLW (xmm, imm8)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

```

VPSRLW (xmm, xmm, xmm/m128)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

VPSRLW (xmm, imm8)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

```

PSRLD (xmm, xmm, xmm/m128)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

PSRLD (xmm, imm8)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

```

VPSRLD (xmm, xmm, xmm/m128)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

VPSRLD (xmm, imm8)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

```

PSRLQ (xmm, xmm, xmm/m128)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

```

PSRLQ (xmm, imm8)

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)

```

DEST[VLMAX-1:128] (Unmodified)

VPSRLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)

DEST[VLMAX-1:128] ← 0

VPSRLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)

DEST[VLMAX-1:128] ← 0

VPSRLW (ymm, ymm, xmm/m128)

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)

VPSRLW (ymm, imm8)

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)

VPSRLD (ymm, ymm, xmm/m128)

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)

VPSRLD (ymm, imm8)

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)

VPSRLQ (ymm, ymm, xmm/m128)

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)

VPSRLQ (ymm, imm8)

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)

Intel C/C++ Compiler Intrinsic Equivalents

PSRLW: __m64 _mm_srli_pi16 (__m64 m, int count)

PSRLW: __m64 _mm_srl_pi16 (__m64 m, __m64 count)

(V)PSRLW: __m128i _mm_srli_epi16 (__m128i m, int count)

(V)PSRLW: __m128i _mm_srl_epi16 (__m128i m, __m128i count)

VPSRLW: __m256i _mm256_srli_epi16 (__m256i m, int count)

VPSRLW: __m256i _mm256_srl_epi16 (__m256i m, __m128i count)

PSRLD: __m64 _mm_srli_pi32 (__m64 m, int count)

PSRLD: __m64 _mm_srl_pi32 (__m64 m, __m64 count)

(V)PSRLD: __m128i _mm_srli_epi32 (__m128i m, int count)

(V)PSRLD: __m128i _mm_srl_epi32 (__m128i m, __m128i count)

VPSRLD: __m256i _mm256_srli_epi32 (__m256i m, int count)

VPSRLD: __m256i _mm256_srl_epi32 (__m256i m, __m128i count)

PSRLQ: __m64 _mm_srli_si64 (__m64 m, int count)

PSRLQ: __m64 _mm_srl_si64 (__m64 m, __m64 count)

(V)PSRLQ: __m128i _mm_srli_epi64 (__m128i m, int count)

(V)PSRLQ: __m128i _mm_srl_epi64 (__m128i m, __m128i count)

VPSRLQ: __m256i _mm256_srli_epi64 (__m256i m, int count)

VPSRLQ: __m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally
#UD If VEX.L = 1.

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F8 /r ¹ PSUBB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract packed byte integers in <i>mm/m64</i> from packed byte integers in <i>mm</i> .
66 0F F8 /r PSUBB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed byte integers in <i>xmm2/m128</i> from packed byte integers in <i>xmm1</i> .
0F F9 /r ¹ PSUBW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract packed word integers in <i>mm/m64</i> from packed word integers in <i>mm</i> .
66 0F F9 /r PSUBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed word integers in <i>xmm2/m128</i> from packed word integers in <i>xmm1</i> .
0F FA /r ¹ PSUBD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract packed doubleword integers in <i>mm/m64</i> from packed doubleword integers in <i>mm</i> .
66 0F FA /r PSUBD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed doubleword integers in <i>xmm2/mem128</i> from packed doubleword integers in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed byte integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed word integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG FA /r VPSUBD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed doubleword integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed byte integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed word integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG FA /r VPSUBD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed doubleword integers in <i>ymm3/m256</i> from <i>ymm2</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBB (with 64-bit operands)

$DEST[7:0] \leftarrow DEST[7:0] - SRC[7:0];$

(* Repeat subtract operation for 2nd through 7th byte *)

$DEST[63:56] \leftarrow DEST[63:56] - SRC[63:56];$

PSUBB (with 128-bit operands)

$DEST[7:0] \leftarrow DEST[7:0] - SRC[7:0];$

(* Repeat subtract operation for 2nd through 14th byte *)

$DEST[127:120] \leftarrow DEST[127:120] - SRC[127:120];$

VPSUBB (VEX.128 encoded version)

$DEST[7:0] \leftarrow SRC1[7:0] - SRC2[7:0]$

$DEST[15:8] \leftarrow SRC1[15:8] - SRC2[15:8]$

$DEST[23:16] \leftarrow SRC1[23:16] - SRC2[23:16]$

$DEST[31:24] \leftarrow SRC1[31:24] - SRC2[31:24]$

$DEST[39:32] \leftarrow SRC1[39:32] - SRC2[39:32]$

$DEST[47:40] \leftarrow SRC1[47:40] - SRC2[47:40]$

$DEST[55:48] \leftarrow SRC1[55:48] - SRC2[55:48]$

$DEST[63:56] \leftarrow SRC1[63:56] - SRC2[63:56]$

$DEST[71:64] \leftarrow SRC1[71:64] - SRC2[71:64]$

$DEST[79:72] \leftarrow SRC1[79:72] - SRC2[79:72]$

$DEST[87:80] \leftarrow SRC1[87:80] - SRC2[87:80]$

$DEST[95:88] \leftarrow SRC1[95:88] - SRC2[95:88]$

$DEST[103:96] \leftarrow SRC1[103:96] - SRC2[103:96]$

$DEST[111:104] \leftarrow SRC1[111:104] - SRC2[111:104]$

$DEST[119:112] \leftarrow SRC1[119:112] - SRC2[119:112]$

$DEST[127:120] \leftarrow SRC1[127:120] - SRC2[127:120]$

$DEST[VLMAX-1:128] \leftarrow 00$

VPSUBB (VEX.256 encoded version)

$\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0] - \text{SRC2}[7:0]$
 $\text{DEST}[15:8] \leftarrow \text{SRC1}[15:8] - \text{SRC2}[15:8]$
 $\text{DEST}[23:16] \leftarrow \text{SRC1}[23:16] - \text{SRC2}[23:16]$
 $\text{DEST}[31:24] \leftarrow \text{SRC1}[31:24] - \text{SRC2}[31:24]$
 $\text{DEST}[39:32] \leftarrow \text{SRC1}[39:32] - \text{SRC2}[39:32]$
 $\text{DEST}[47:40] \leftarrow \text{SRC1}[47:40] - \text{SRC2}[47:40]$
 $\text{DEST}[55:48] \leftarrow \text{SRC1}[55:48] - \text{SRC2}[55:48]$
 $\text{DEST}[63:56] \leftarrow \text{SRC1}[63:56] - \text{SRC2}[63:56]$
 $\text{DEST}[71:64] \leftarrow \text{SRC1}[71:64] - \text{SRC2}[71:64]$
 $\text{DEST}[79:72] \leftarrow \text{SRC1}[79:72] - \text{SRC2}[79:72]$
 $\text{DEST}[87:80] \leftarrow \text{SRC1}[87:80] - \text{SRC2}[87:80]$
 $\text{DEST}[95:88] \leftarrow \text{SRC1}[95:88] - \text{SRC2}[95:88]$
 $\text{DEST}[103:96] \leftarrow \text{SRC1}[103:96] - \text{SRC2}[103:96]$
 $\text{DEST}[111:104] \leftarrow \text{SRC1}[111:104] - \text{SRC2}[111:104]$
 $\text{DEST}[119:112] \leftarrow \text{SRC1}[119:112] - \text{SRC2}[119:112]$
 $\text{DEST}[127:120] \leftarrow \text{SRC1}[127:120] - \text{SRC2}[127:120]$
 $\text{DEST}[135:128] \leftarrow \text{SRC1}[135:128] - \text{SRC2}[135:128]$
 $\text{DEST}[143:136] \leftarrow \text{SRC1}[143:136] - \text{SRC2}[143:136]$
 $\text{DEST}[151:144] \leftarrow \text{SRC1}[151:144] - \text{SRC2}[151:144]$
 $\text{DEST}[159:152] \leftarrow \text{SRC1}[159:152] - \text{SRC2}[159:152]$
 $\text{DEST}[167:160] \leftarrow \text{SRC1}[167:160] - \text{SRC2}[167:160]$
 $\text{DEST}[175:168] \leftarrow \text{SRC1}[175:168] - \text{SRC2}[175:168]$
 $\text{DEST}[183:176] \leftarrow \text{SRC1}[183:176] - \text{SRC2}[183:176]$
 $\text{DEST}[191:184] \leftarrow \text{SRC1}[191:184] - \text{SRC2}[191:184]$
 $\text{DEST}[199:192] \leftarrow \text{SRC1}[199:192] - \text{SRC2}[199:192]$
 $\text{DEST}[207:200] \leftarrow \text{SRC1}[207:200] - \text{SRC2}[207:200]$
 $\text{DEST}[215:208] \leftarrow \text{SRC1}[215:208] - \text{SRC2}[215:208]$
 $\text{DEST}[223:216] \leftarrow \text{SRC1}[223:216] - \text{SRC2}[223:216]$
 $\text{DEST}[231:224] \leftarrow \text{SRC1}[231:224] - \text{SRC2}[231:224]$
 $\text{DEST}[239:232] \leftarrow \text{SRC1}[239:232] - \text{SRC2}[239:232]$
 $\text{DEST}[247:240] \leftarrow \text{SRC1}[247:240] - \text{SRC2}[247:240]$
 $\text{DEST}[255:248] \leftarrow \text{SRC1}[255:248] - \text{SRC2}[255:248]$

PSUBW (with 64-bit operands)

$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0] - \text{SRC}[15:0];$
 (* Repeat subtract operation for 2nd and 3rd word *)
 $\text{DEST}[63:48] \leftarrow \text{DEST}[63:48] - \text{SRC}[63:48];$

PSUBW (with 128-bit operands)

$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0] - \text{SRC}[15:0];$
 (* Repeat subtract operation for 2nd through 7th word *)
 $\text{DEST}[127:112] \leftarrow \text{DEST}[127:112] - \text{SRC}[127:112];$

VPSUBW (VEX.128 encoded version)

$\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] - \text{SRC2}[15:0]$
 $\text{DEST}[31:16] \leftarrow \text{SRC1}[31:16] - \text{SRC2}[31:16]$
 $\text{DEST}[47:32] \leftarrow \text{SRC1}[47:32] - \text{SRC2}[47:32]$
 $\text{DEST}[63:48] \leftarrow \text{SRC1}[63:48] - \text{SRC2}[63:48]$
 $\text{DEST}[79:64] \leftarrow \text{SRC1}[79:64] - \text{SRC2}[79:64]$
 $\text{DEST}[95:80] \leftarrow \text{SRC1}[95:80] - \text{SRC2}[95:80]$
 $\text{DEST}[111:96] \leftarrow \text{SRC1}[111:96] - \text{SRC2}[111:96]$
 $\text{DEST}[127:112] \leftarrow \text{SRC1}[127:112] - \text{SRC2}[127:112]$

DEST[VLMAX-1:128] \leftarrow 0

VPSUBW (VEX.256 encoded version)

DEST[15:0] \leftarrow SRC1[15:0]-SRC2[15:0]
 DEST[31:16] \leftarrow SRC1[31:16]-SRC2[31:16]
 DEST[47:32] \leftarrow SRC1[47:32]-SRC2[47:32]
 DEST[63:48] \leftarrow SRC1[63:48]-SRC2[63:48]
 DEST[79:64] \leftarrow SRC1[79:64]-SRC2[79:64]
 DEST[95:80] \leftarrow SRC1[95:80]-SRC2[95:80]
 DEST[111:96] \leftarrow SRC1[111:96]-SRC2[111:96]
 DEST[127:112] \leftarrow SRC1[127:112]-SRC2[127:112]
 DEST[143:128] \leftarrow SRC1[143:128]-SRC2[143:128]
 DEST[159:144] \leftarrow SRC1[159:144]-SRC2[159:144]
 DEST[175:160] \leftarrow SRC1[175:160]-SRC2[175:160]
 DEST[191:176] \leftarrow SRC1[191:176]-SRC2[191:176]
 DEST[207:192] \leftarrow SRC1[207:192]-SRC2[207:192]
 DEST[223:208] \leftarrow SRC1[223:208]-SRC2[223:208]
 DEST[239:224] \leftarrow SRC1[239:224]-SRC2[239:224]
 DEST[255:240] \leftarrow SRC1[255:240]-SRC2[255:240]

PSUBD (with 64-bit operands)

DEST[31:0] \leftarrow DEST[31:0] – SRC[31:0];
 DEST[63:32] \leftarrow DEST[63:32] – SRC[63:32];

PSUBD (with 128-bit operands)

DEST[31:0] \leftarrow DEST[31:0] – SRC[31:0];
 (* Repeat subtract operation for 2nd and 3rd doubleword *)
 DEST[127:96] \leftarrow DEST[127:96] – SRC[127:96];

VPSUBD (VEX.128 encoded version)

DEST[31:0] \leftarrow SRC1[31:0]-SRC2[31:0]
 DEST[63:32] \leftarrow SRC1[63:32]-SRC2[63:32]
 DEST[95:64] \leftarrow SRC1[95:64]-SRC2[95:64]
 DEST[127:96] \leftarrow SRC1[127:96]-SRC2[127:96]
 DEST[VLMAX-1:128] \leftarrow 0

VPSUBD (VEX.256 encoded version)

DEST[31:0] \leftarrow SRC1[31:0]-SRC2[31:0]
 DEST[63:32] \leftarrow SRC1[63:32]-SRC2[63:32]
 DEST[95:64] \leftarrow SRC1[95:64]-SRC2[95:64]
 DEST[127:96] \leftarrow SRC1[127:96]-SRC2[127:96]
 DEST[159:128] \leftarrow SRC1[159:128]-SRC2[159:128]
 DEST[191:160] \leftarrow SRC1[191:160]-SRC2[191:160]
 DEST[223:192] \leftarrow SRC1[223:192]-SRC2[223:192]
 DEST[255:224] \leftarrow SRC1[255:224]-SRC2[255:224]

Intel C/C++ Compiler Intrinsic Equivalents

PSUBB: `__m64 _mm_sub_pi8(__m64 m1, __m64 m2)`
 (V)PSUBB: `__m128i _mm_sub_epi8 (__m128i a, __m128i b)`
 VPSUBB: `__m256i _mm256_sub_epi8 (__m256i a, __m256i b)`
 PSUBW: `__m64 _mm_sub_pi16(__m64 m1, __m64 m2)`
 (V)PSUBW: `__m128i _mm_sub_epi16 (__m128i a, __m128i b)`
 VPSUBW: `__m256i _mm256_sub_epi16 (__m256i a, __m256i b)`

PSUBD: __m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD: __m128i _mm_sub_epi32 (__m128i a, __m128i b)
VPSUBD: __m256i _mm256_sub_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FB /r ¹ PSUBQ mm1, mm2/m64	RM	V/V	SSE2	Subtract quadword integer in mm1 from mm2 /m64.
66 0F FB /r PSUBQ xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed quadword integers in xmm1 from xmm2 /m128.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.NDS.256.66.0F.WIG FB/r VPSUBQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBQ (with 64-Bit operands)

DEST[63:0] ← DEST[63:0] – SRC[63:0];

PSUBQ (with 128-Bit operands)

DEST[63:0] \leftarrow DEST[63:0] – SRC[63:0];
 DEST[127:64] \leftarrow DEST[127:64] – SRC[127:64];

VPSUBQ (VEX.128 encoded version)

DEST[63:0] \leftarrow SRC1[63:0]-SRC2[63:0]
 DEST[127:64] \leftarrow SRC1[127:64]-SRC2[127:64]
 DEST[VLMAX-1:128] \leftarrow 0

VPSUBQ (VEX.256 encoded version)

DEST[63:0] \leftarrow SRC1[63:0]-SRC2[63:0]
 DEST[127:64] \leftarrow SRC1[127:64]-SRC2[127:64]
 DEST[191:128] \leftarrow SRC1[191:128]-SRC2[191:128]
 DEST[255:192] \leftarrow SRC1[255:192]-SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalents

PSUBQ: __m64 _mm_sub_si64(__m64 m1, __m64 m2)
 (V)PSUBQ: __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
 VPSUBQ: __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E8 /r ¹ PSUBSB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract signed packed bytes in <i>mm/m64</i> from signed packed bytes in <i>mm</i> and saturate results.
66 0F E8 /r PSUBSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed signed byte integers in <i>xmm2/m128</i> from packed signed byte integers in <i>xmm1</i> and saturate results.
0F E9 /r ¹ PSUBSW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract signed packed words in <i>mm/m64</i> from signed packed words in <i>mm</i> and saturate results.
66 0F E9 /r PSUBSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed signed word integers in <i>xmm2/m128</i> from packed signed word integers in <i>xmm1</i> and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed signed byte integers in <i>xmm3/m128</i> from packed signed byte integers in <i>xmm2</i> and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed signed word integers in <i>xmm3/m128</i> from packed signed word integers in <i>xmm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed signed byte integers in <i>ymm3/m256</i> from packed signed byte integers in <i>ymm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed signed word integers in <i>ymm3/m256</i> from packed signed word integers in <i>ymm2</i> and saturate results.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56]);
```

PSUBSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] – SRC[127:120]);
```

VPSUBSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

VPSUBSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
```

PSUBSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat subtract operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48]);
```

PSUBSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] – SRC[127:112]);
```

VPSUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```

VPSUBSW (VEX.256 encoded version)

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);

(* Repeat subtract operation for 2nd through 15th words *)

DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);

Intel C/C++ Compiler Intrinsic Equivalents

PSUBSB: __m64 _mm_subs_pi8(__m64 m1, __m64 m2)

(V)PSUBSB: __m128i _mm_subs_epi8(__m128i m1, __m128i m2)

VPSUBSB: __m256i _mm256_subs_epi8(__m256i m1, __m256i m2)

PSUBSW: __m64 _mm_subs_pi16(__m64 m1, __m64 m2)

(V)PSUBSW: __m128i _mm_subs_epi16(__m128i m1, __m128i m2)

VPSUBSW: __m256i _mm256_subs_epi16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D8 /r ¹ PSUBUSB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract unsigned packed bytes in <i>mm/m64</i> from unsigned packed bytes in <i>mm</i> and saturate result.
66 0F D8 /r PSUBUSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed unsigned byte integers in <i>xmm2/m128</i> from packed unsigned byte integers in <i>xmm1</i> and saturate result.
0F D9 /r ¹ PSUBUSW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract unsigned packed words in <i>mm/m64</i> from unsigned packed words in <i>mm</i> and saturate result.
66 0F D9 /r PSUBUSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed unsigned word integers in <i>xmm2/m128</i> from packed unsigned word integers in <i>xmm1</i> and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed unsigned byte integers in <i>xmm3/m128</i> from packed unsigned byte integers in <i>xmm2</i> and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed unsigned word integers in <i>xmm3/m128</i> from packed unsigned word integers in <i>xmm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed unsigned byte integers in <i>ymm3/m256</i> from packed unsigned byte integers in <i>ymm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed unsigned word integers in <i>ymm3/m256</i> from packed unsigned word integers in <i>ymm2</i> and saturate result.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56]);
```

PSUBUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] – SRC[127:120]);
```

VPSUBUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

VPSUBUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
```

PSUBUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48]);
```

PSUBUSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] – SRC[127:112]);
```

VPSUBUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[VLMAX-1:128] ← 0
```


VPSUBUSB (VEX.256 encoded version)

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);

(* Repeat subtract operation for 2nd through 15th words *)

DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);

Intel C/C++ Compiler Intrinsic Equivalents

PSUBUSB: __m64 _mm_subs_pu8(__m64 m1, __m64 m2)

(V)PSUBUSB: __m128i _mm_subs_epu8(__m128i m1, __m128i m2)

VPSUBUSB: __m256i _mm256_subs_epu8(__m256i m1, __m256i m2)

PSUBUSW: __m64 _mm_subs_pu16(__m64 m1, __m64 m2)

(V)PSUBUSW: __m128i _mm_subs_epu16(__m128i m1, __m128i m2)

VPSUBUSW: __m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PTEST- Logical Compare

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Set ZF if <i>xmm2/m128</i> AND <i>xmm1</i> result is all 0s. Set CF if <i>xmm2/m128</i> AND NOT <i>xmm1</i> result is all 0s.
VEX.128.66.0F38.WIG 17 /r VPTEST <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.
VEX.256.66.0F38.WIG 17 /r VPTEST <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)PTEST (128-bit version)

IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)

THEN ZF ← 1;

ELSE ZF ← 0;

IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)

THEN CF ← 1;

ELSE CF ← 0;

DEST (unmodified)

AF ← OF ← PF ← SF ← 0;

VPTEST (VEX.256 encoded version)

IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF ← 1;

ELSE ZF ← 0;

IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF ← 1;

ELSE CF ← 0;

DEST (unmodified)

AF ← OF ← PF ← SF ← 0;

Intel C/C++ Compiler Intrinsic Equivalent

PTEST

```
int _mm_testz_si128 (__m128i s1, __m128i s2);  
int _mm_testc_si128 (__m128i s1, __m128i s2);  
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

VPTEST

```
int _mm256_testz_si256 (__m256i s1, __m256i s2);  
int _mm256_testc_si256 (__m256i s1, __m256i s2);  
int _mm256_testnzc_si256 (__m256i s1, __m256i s2);  
int _mm_testz_si128 (__m128i s1, __m128i s2);  
int _mm_testc_si128 (__m128i s1, __m128i s2);  
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

Flags Affected

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 68 /r ¹ PUNPCKHBW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 OF 68 /r PUNPCKHBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
OF 69 /r ¹ PUNPCKHWD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 OF 69 /r PUNPCKHWD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
OF 6A /r ¹ PUNPCKHDQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 OF 6A /r PUNPCKHDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 OF 6D /r PUNPCKHQDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 68/r VPUNPCKHBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 69/r VPUNPCKHWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 6A/r VPUNPCKHDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 6D/r VPUNPCKHQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.OF.WIG 68 /r VPUNPCKHBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 69 /r VPUNPCKHWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 6A /r VPUNPCKHDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 6D /r VPUNPCKHQDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-16 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

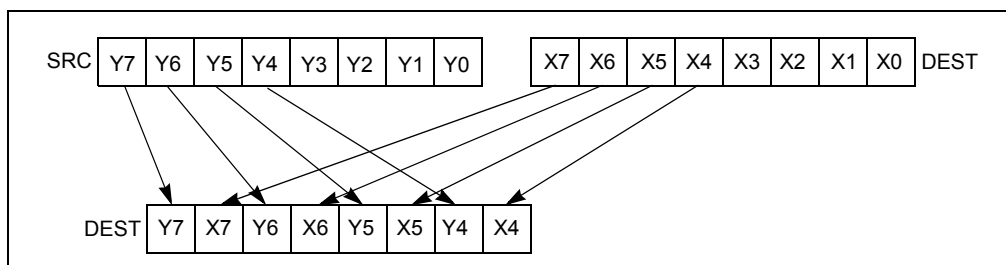


Figure 4-16. PUNPCKHBW Instruction Operation Using 64-bit Operands

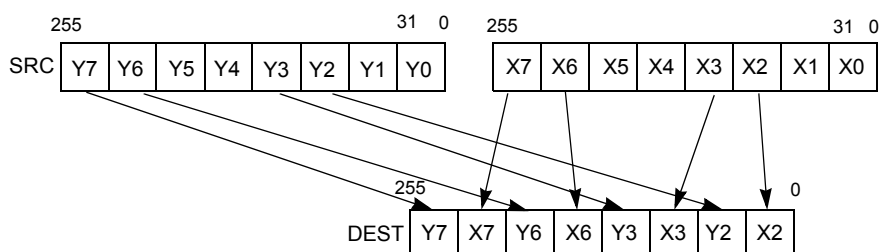


Figure 4-17. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

PUNPCKHBW instruction with 128-bit operands:

```
DEST[7:0] ← DEST[71:64];
DEST[15:8] ← SRC[71:64];
DEST[23:16] ← DEST[79:72];
DEST[31:24] ← SRC[79:72];
DEST[39:32] ← DEST[87:80];
DEST[47:40] ← SRC[87:80];
DEST[55:48] ← DEST[95:88];
DEST[63:56] ← SRC[95:88];
DEST[71:64] ← DEST[103:96];
DEST[79:72] ← SRC[103:96];
DEST[87:80] ← DEST[111:104];
DEST[95:88] ← SRC[111:104];
DEST[103:96] ← DEST[119:112];
DEST[111:104] ← SRC[119:112];
DEST[119:112] ← DEST[127:120];
DEST[127:120] ← SRC[127:120];
```

PUNPCKHWD instruction with 128-bit operands:

```
DEST[15:0] ← DEST[79:64];
DEST[31:16] ← SRC[79:64];
DEST[47:32] ← DEST[95:80];
DEST[63:48] ← SRC[95:80];
DEST[79:64] ← DEST[111:96];
DEST[95:80] ← SRC[111:96];
DEST[111:96] ← DEST[127:112];
```

DEST[127:112] \leftarrow SRC[127:112];

PUNPCKHDQ instruction with 128-bit operands:

DEST[31:0] \leftarrow DEST[95:64];
 DEST[63:32] \leftarrow SRC[95:64];
 DEST[95:64] \leftarrow DEST[127:96];
 DEST[127:96] \leftarrow SRC[127:96];

PUNPCKHQDQ instruction:

DEST[63:0] \leftarrow DEST[127:64];
 DEST[127:64] \leftarrow SRC[127:64];

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)

DEST[7:0] \leftarrow SRC1[71:64]
 DEST[15:8] \leftarrow SRC2[71:64]
 DEST[23:16] \leftarrow SRC1[79:72]
 DEST[31:24] \leftarrow SRC2[79:72]
 DEST[39:32] \leftarrow SRC1[87:80]
 DEST[47:40] \leftarrow SRC2[87:80]
 DEST[55:48] \leftarrow SRC1[95:88]
 DEST[63:56] \leftarrow SRC2[95:88]
 DEST[71:64] \leftarrow SRC1[103:96]
 DEST[79:72] \leftarrow SRC2[103:96]
 DEST[87:80] \leftarrow SRC1[111:104]
 DEST[95:88] \leftarrow SRC2[111:104]
 DEST[103:96] \leftarrow SRC1[119:112]
 DEST[111:104] \leftarrow SRC2[119:112]
 DEST[119:112] \leftarrow SRC1[127:120]
 DEST[127:120] \leftarrow SRC2[127:120]
 DEST[135:128] \leftarrow SRC1[199:192]
 DEST[143:136] \leftarrow SRC2[199:192]
 DEST[151:144] \leftarrow SRC1[207:200]
 DEST[159:152] \leftarrow SRC2[207:200]
 DEST[167:160] \leftarrow SRC1[215:208]
 DEST[175:168] \leftarrow SRC2[215:208]
 DEST[183:176] \leftarrow SRC1[223:216]
 DEST[191:184] \leftarrow SRC2[223:216]
 DEST[199:192] \leftarrow SRC1[231:224]
 DEST[207:200] \leftarrow SRC2[231:224]
 DEST[215:208] \leftarrow SRC1[239:232]
 DEST[223:216] \leftarrow SRC2[239:232]
 DEST[231:224] \leftarrow SRC1[247:240]
 DEST[239:232] \leftarrow SRC2[247:240]
 DEST[247:240] \leftarrow SRC1[255:248]
 DEST[255:248] \leftarrow SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)

DEST[7:0] \leftarrow SRC1[71:64]
 DEST[15:8] \leftarrow SRC2[71:64]
 DEST[23:16] \leftarrow SRC1[79:72]
 DEST[31:24] \leftarrow SRC2[79:72]
 DEST[39:32] \leftarrow SRC1[87:80]
 DEST[47:40] \leftarrow SRC2[87:80]
 DEST[55:48] \leftarrow SRC1[95:88]
 DEST[63:56] \leftarrow SRC2[95:88]

DEST[71:64] ← SRC1[103:96]
 DEST[79:72] ← SRC2[103:96]
 DEST[87:80] ← SRC1[111:104]
 DEST[95:88] ← SRC2[111:104]
 DEST[103:96] ← SRC1[119:112]
 DEST[111:104] ← SRC2[119:112]
 DEST[119:112] ← SRC1[127:120]
 DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]
 DEST[31:16] ← SRC2[79:64]
 DEST[47:32] ← SRC1[95:80]
 DEST[63:48] ← SRC2[95:80]
 DEST[79:64] ← SRC1[111:96]
 DEST[95:80] ← SRC2[111:96]
 DEST[111:96] ← SRC1[127:112]
 DEST[127:112] ← SRC2[127:112]
 DEST[143:128] ← SRC1[207:192]
 DEST[159:144] ← SRC2[207:192]
 DEST[175:160] ← SRC1[223:208]
 DEST[191:176] ← SRC2[223:208]
 DEST[207:192] ← SRC1[239:224]
 DEST[223:208] ← SRC2[239:224]
 DEST[239:224] ← SRC1[255:240]
 DEST[255:240] ← SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]
 DEST[31:16] ← SRC2[79:64]
 DEST[47:32] ← SRC1[95:80]
 DEST[63:48] ← SRC2[95:80]
 DEST[79:64] ← SRC1[111:96]
 DEST[95:80] ← SRC2[111:96]
 DEST[111:96] ← SRC1[127:112]
 DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)

DEST[31:0] ← SRC1[95:64]
 DEST[63:32] ← SRC2[95:64]
 DEST[95:64] ← SRC1[127:96]
 DEST[127:96] ← SRC2[127:96]
 DEST[159:128] ← SRC1[223:192]
 DEST[191:160] ← SRC2[223:192]
 DEST[223:192] ← SRC1[255:224]
 DEST[255:224] ← SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)

DEST[31:0] ← SRC1[95:64]
 DEST[63:32] ← SRC2[95:64]
 DEST[95:64] ← SRC1[127:96]
 DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$
 $\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[255:192]$
 $\text{DEST}[255:192] \leftarrow \text{SRC2}[255:192]$

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$
 $\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$

PUNPCKHBW (128-bit Legacy SSE Version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_BYTES}(\text{DEST}, \text{SRC})$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VPUNPCKHBW (VEX.128 encoded version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_BYTES}(\text{SRC1}, \text{SRC2})$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPUNPCKHBW (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE_HIGH_BYTES_256b}(\text{SRC1}, \text{SRC2})$

PUNPCKHWD (128-bit Legacy SSE Version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_WORDS}(\text{DEST}, \text{SRC})$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VPUNPCKHWD (VEX.128 encoded version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_WORDS}(\text{SRC1}, \text{SRC2})$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPUNPCKHWD (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE_HIGH_WORDS_256b}(\text{SRC1}, \text{SRC2})$

PUNPCKHDQ (128-bit Legacy SSE Version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_DWORDS}(\text{DEST}, \text{SRC})$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VPUNPCKHDQ (VEX.128 encoded version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_DWORDS}(\text{SRC1}, \text{SRC2})$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VPUNPCKHDQ (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE_HIGH_DWORDS_256b}(\text{SRC1}, \text{SRC2})$

PUNPCKHQDQ (128-bit Legacy SSE Version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_QWORDS}(\text{DEST}, \text{SRC})$
 $\text{DEST}[255:127]$ (Unmodified)

VPUNPCKHQDQ (VEX.128 encoded version)

$\text{DEST}[127:0] \leftarrow \text{INTERLEAVE_HIGH_QWORDS}(\text{SRC1}, \text{SRC2})$
 $\text{DEST}[255:127] \leftarrow 0$

VPUNPCKHQDQ (VEX.256 encoded version)

$\text{DEST}[255:0] \leftarrow \text{INTERLEAVE_HIGH_QWORDS_256}(\text{SRC1}, \text{SRC2})$

Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKHBW: __m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
 (V)PUNPCKHBW: __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
 VPUNPCKHBW: __m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
 PUNPCKHWD: __m64 _mm_unpackhi_pi16(__m64 m1, __m64 m2)
 (V)PUNPCKHWD: __m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)
 VPUNPCKHWD: __m256i _mm256_unpackhi_epi16(__m256i m1, __m256i m2)
 PUNPCKHDQ: __m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
 (V)PUNPCKHDQ: __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
 VPUNPCKHDQ: __m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
 (V)PUNPCKHQDQ: __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
 VPUNPCKHQDQ: __m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 60 /r ¹ PUNPCKLBW <i>mm</i> , <i>mm/m32</i>	RM	V/V	MMX	Interleave low-order bytes from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 0F 60 /r PUNPCKLBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 61 /r ¹ PUNPCKLWD <i>mm</i> , <i>mm/m32</i>	RM	V/V	MMX	Interleave low-order words from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 0F 61 /r PUNPCKLWD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 62 /r ¹ PUNPCKLDQ <i>mm</i> , <i>mm/m32</i>	RM	V/V	MMX	Interleave low-order doublewords from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 0F 62 /r PUNPCKLDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6C /r PUNPCKLQDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order quadword from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.0F.WIG 60/r VPUNPCKLBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 61/r VPUNPCKLWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 62/r VPUNPCKLDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 6C/r VPUNPCKLQDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-18 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

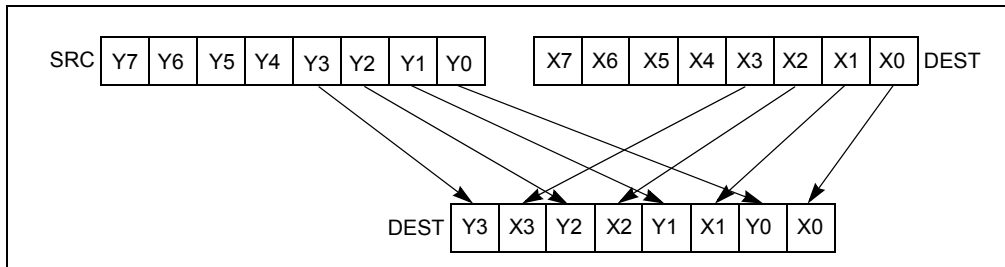


Figure 4-18. PUNPCKLBW Instruction Operation Using 64-bit Operands

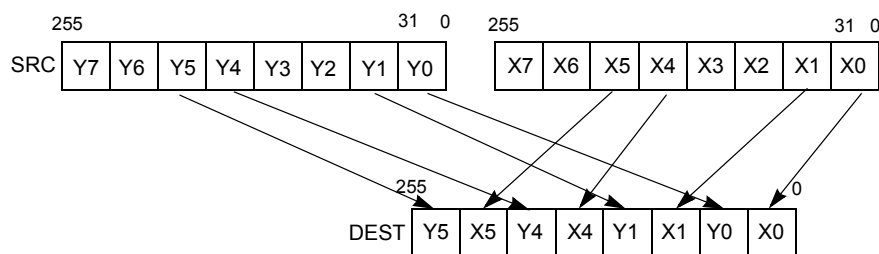


Figure 4-19. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

PUNPCKLBW instruction with 128-bit operands:

```
DEST[7:0] ← DEST[7:0];
DEST[15:8] ← SRC[7:0];
DEST[23:16] ← DEST[15:8];
DEST[31:24] ← SRC[15:8];
DEST[39:32] ← DEST[23:16];
DEST[47:40] ← SRC[23:16];
DEST[55:48] ← DEST[31:24];
DEST[63:56] ← SRC[31:24];
DEST[71:64] ← DEST[39:32];
DEST[79:72] ← SRC[39:32];
DEST[87:80] ← DEST[47:40];
DEST[95:88] ← SRC[47:40];
DEST[103:96] ← DEST[55:48];
DEST[111:104] ← SRC[55:48];
DEST[119:112] ← DEST[63:56];
DEST[127:120] ← SRC[63:56];
```

PUNPCKLWD instruction with 128-bit operands:

```
DEST[15:0] ← DEST[15:0];
DEST[31:16] ← SRC[15:0];
DEST[47:32] ← DEST[31:16];
DEST[63:48] ← SRC[31:16];
DEST[79:64] ← DEST[47:32];
DEST[95:80] ← SRC[47:32];
DEST[111:96] ← DEST[63:48];
DEST[127:112] ← SRC[63:48];
```

PUNPCKLDQ instruction with 128-bit operands:

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0];$
 $\text{DEST}[63:32] \leftarrow \text{SRC}[31:0];$
 $\text{DEST}[95:64] \leftarrow \text{DEST}[63:32];$
 $\text{DEST}[127:96] \leftarrow \text{SRC}[63:32];$

PUNPCKLQDQ

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0];$
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0];$

INTERLEAVE_BYTES_256b (SRC1, SRC2)

$\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0]$
 $\text{DEST}[15:8] \leftarrow \text{SRC2}[7:0]$
 $\text{DEST}[23:16] \leftarrow \text{SRC1}[15:8]$
 $\text{DEST}[31:24] \leftarrow \text{SRC2}[15:8]$
 $\text{DEST}[39:32] \leftarrow \text{SRC1}[23:16]$
 $\text{DEST}[47:40] \leftarrow \text{SRC2}[23:16]$
 $\text{DEST}[55:48] \leftarrow \text{SRC1}[31:24]$
 $\text{DEST}[63:56] \leftarrow \text{SRC2}[31:24]$
 $\text{DEST}[71:64] \leftarrow \text{SRC1}[39:32]$
 $\text{DEST}[79:72] \leftarrow \text{SRC2}[39:32]$
 $\text{DEST}[87:80] \leftarrow \text{SRC1}[47:40]$
 $\text{DEST}[95:88] \leftarrow \text{SRC2}[47:40]$
 $\text{DEST}[103:96] \leftarrow \text{SRC1}[55:48]$
 $\text{DEST}[111:104] \leftarrow \text{SRC2}[55:48]$
 $\text{DEST}[119:112] \leftarrow \text{SRC1}[63:56]$
 $\text{DEST}[127:120] \leftarrow \text{SRC2}[63:56]$
 $\text{DEST}[135:128] \leftarrow \text{SRC1}[135:128]$
 $\text{DEST}[143:136] \leftarrow \text{SRC2}[135:128]$
 $\text{DEST}[151:144] \leftarrow \text{SRC1}[143:136]$
 $\text{DEST}[159:152] \leftarrow \text{SRC2}[143:136]$
 $\text{DEST}[167:160] \leftarrow \text{SRC1}[151:144]$
 $\text{DEST}[175:168] \leftarrow \text{SRC2}[151:144]$
 $\text{DEST}[183:176] \leftarrow \text{SRC1}[159:152]$
 $\text{DEST}[191:184] \leftarrow \text{SRC2}[159:152]$
 $\text{DEST}[199:192] \leftarrow \text{SRC1}[167:160]$
 $\text{DEST}[207:200] \leftarrow \text{SRC2}[167:160]$
 $\text{DEST}[215:208] \leftarrow \text{SRC1}[175:168]$
 $\text{DEST}[223:216] \leftarrow \text{SRC2}[175:168]$
 $\text{DEST}[231:224] \leftarrow \text{SRC1}[183:176]$
 $\text{DEST}[239:232] \leftarrow \text{SRC2}[183:176]$
 $\text{DEST}[247:240] \leftarrow \text{SRC1}[191:184]$
 $\text{DEST}[255:248] \leftarrow \text{SRC2}[191:184]$

INTERLEAVE_BYTES (SRC1, SRC2)

$\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0]$
 $\text{DEST}[15:8] \leftarrow \text{SRC2}[7:0]$
 $\text{DEST}[23:16] \leftarrow \text{SRC2}[15:8]$
 $\text{DEST}[31:24] \leftarrow \text{SRC2}[15:8]$
 $\text{DEST}[39:32] \leftarrow \text{SRC1}[23:16]$
 $\text{DEST}[47:40] \leftarrow \text{SRC2}[23:16]$
 $\text{DEST}[55:48] \leftarrow \text{SRC1}[31:24]$
 $\text{DEST}[63:56] \leftarrow \text{SRC2}[31:24]$
 $\text{DEST}[71:64] \leftarrow \text{SRC1}[39:32]$
 $\text{DEST}[79:72] \leftarrow \text{SRC2}[39:32]$

DEST[87:80] ← SRC1[47:40]
 DEST[95:88] ← SRC2[47:40]
 DEST[103:96] ← SRC1[55:48]
 DEST[111:104] ← SRC2[55:48]
 DEST[119:112] ← SRC1[63:56]
 DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[15:0]
 DEST[31:16] ← SRC2[15:0]
 DEST[47:32] ← SRC1[31:16]
 DEST[63:48] ← SRC2[31:16]
 DEST[79:64] ← SRC1[47:32]
 DEST[95:80] ← SRC2[47:32]
 DEST[111:96] ← SRC1[63:48]
 DEST[127:112] ← SRC2[63:48]
 DEST[143:128] ← SRC1[143:128]
 DEST[159:144] ← SRC2[143:128]
 DEST[175:160] ← SRC1[159:144]
 DEST[191:176] ← SRC2[159:144]
 DEST[207:192] ← SRC1[175:160]
 DEST[223:208] ← SRC2[175:160]
 DEST[239:224] ← SRC1[191:176]
 DEST[255:240] ← SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[15:0]
 DEST[31:16] ← SRC2[15:0]
 DEST[47:32] ← SRC1[31:16]
 DEST[63:48] ← SRC2[31:16]
 DEST[79:64] ← SRC1[47:32]
 DEST[95:80] ← SRC2[47:32]
 DEST[111:96] ← SRC1[63:48]
 DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS_256b(SRC1, SRC2)

DEST[31:0] ← SRC1[31:0]
 DEST[63:32] ← SRC2[31:0]
 DEST[95:64] ← SRC1[63:32]
 DEST[127:96] ← SRC2[63:32]
 DEST[159:128] ← SRC1[159:128]
 DEST[191:160] ← SRC2[159:128]
 DEST[223:192] ← SRC1[191:160]
 DEST[255:224] ← SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)

DEST[31:0] ← SRC1[31:0]
 DEST[63:32] ← SRC2[31:0]
 DEST[95:64] ← SRC1[63:32]
 DEST[127:96] ← SRC2[63:32]

INTERLEAVE_QWORDS_256b(SRC1, SRC2)

DEST[63:0] ← SRC1[63:0]
 DEST[127:64] ← SRC2[63:0]

DEST[191:128] \leftarrow SRC1[191:128]
 DEST[255:192] \leftarrow SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)

DEST[63:0] \leftarrow SRC1[63:0]
 DEST[127:64] \leftarrow SRC2[63:0]

PUNPCKLBW (128-bit Legacy SSE Version)

DEST[127:0] \leftarrow INTERLEAVE_BYTES(DEST, SRC)
 DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)

DEST[127:0] \leftarrow INTERLEAVE_BYTES(SRC1, SRC2)
 DEST[VLMAX-1:128] \leftarrow 0

VPUNPCKLBW (VEX.256 encoded instruction)

DEST[255:0] \leftarrow INTERLEAVE_BYTES_128b(SRC1, SRC2)

PUNPCKLWD (128-bit Legacy SSE Version)

DEST[127:0] \leftarrow INTERLEAVE_WORDS(DEST, SRC)
 DEST[255:127] (Unmodified)

VPUNPCKLWD (VEX.128 encoded instruction)

DEST[127:0] \leftarrow INTERLEAVE_WORDS(SRC1, SRC2)
 DEST[VLMAX-1:128] \leftarrow 0

VPUNPCKLWD (VEX.256 encoded instruction)

DEST[255:0] \leftarrow INTERLEAVE_WORDS(SRC1, SRC2)

PUNPCKLDQ (128-bit Legacy SSE Version)

DEST[127:0] \leftarrow INTERLEAVE_DWORDS(DEST, SRC)
 DEST[255:127] (Unmodified)

VPUNPCKLDQ (VEX.128 encoded instruction)

DEST[127:0] \leftarrow INTERLEAVE_DWORDS(SRC1, SRC2)
 DEST[VLMAX-1:128] \leftarrow 0

VPUNPCKLDQ (VEX.256 encoded instruction)

DEST[255:0] \leftarrow INTERLEAVE_DWORDS(SRC1, SRC2)

PUNPCKLQDQ (128-bit Legacy SSE Version)

DEST[127:0] \leftarrow INTERLEAVE_QWORDS(DEST, SRC)
 DEST[255:127] (Unmodified)

VPUNPCKLQDQ (VEX.128 encoded instruction)

DEST[127:0] \leftarrow INTERLEAVE_QWORDS(SRC1, SRC2)
 DEST[VLMAX-1:128] \leftarrow 0

VPUNPCKLQDQ (VEX.256 encoded instruction)

DEST[255:0] \leftarrow INTERLEAVE_QWORDS(SRC1, SRC2)

Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKLBW: `__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)`

(V)PUNPCKLBW: __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
 VPUNPCKLBW: __m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
 PUNPCKLWD: __m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
 (V)PUNPCKLWD: __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
 VPUNPCKLWD: __m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
 PUNPCKLDQ: __m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
 (V)PUNPCKLDQ: __m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
 VPUNPCKLDQ: __m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
 (V)PUNPCKLQDQ: __m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
 VPUNPCKLQDQ: __m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH <i>r/m16</i>	M	Valid	Valid	Push <i>r/m16</i> .
FF /6	PUSH <i>r/m32</i>	M	N.E.	Valid	Push <i>r/m32</i> .
FF /6	PUSH <i>r/m64</i>	M	Valid	N.E.	Push <i>r/m64</i> .
50+ <i>rw</i>	PUSH <i>r16</i>	O	Valid	Valid	Push <i>r16</i> .
50+ <i>rd</i>	PUSH <i>r32</i>	O	N.E.	Valid	Push <i>r32</i> .
50+ <i>rd</i>	PUSH <i>r64</i>	O	Valid	N.E.	Push <i>r64</i> .
6A <i>ib</i>	PUSH <i>imm8</i>	I	Valid	Valid	Push <i>imm8</i> .
68 <i>iw</i>	PUSH <i>imm16</i>	I	Valid	Valid	Push <i>imm16</i> .
68 <i>id</i>	PUSH <i>imm32</i>	I	Valid	Valid	Push <i>imm32</i> .
0E	PUSH CS	NP	Invalid	Valid	Push CS.
16	PUSH SS	NP	Invalid	Valid	Push SS.
1E	PUSH DS	NP	Invalid	Valid	Push DS.
06	PUSH ES	NP	Invalid	Valid	Push ES.
0F A0	PUSH FS	NP	Valid	Valid	Push FS.
0F A8	PUSH GS	NP	Valid	Valid	Push GS.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
NP	NA	NA	NA	NA

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
The address size is used only when referencing a source operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

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

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

(* See Description section for possible sign-extension or zero-extension of source operand and for *)

(* a case in which the size of the memory store may be smaller than the instruction's operand size *)

IF StackAddrSize = 64

THEN

IF OperandSize = 64

THEN

RSP ← RSP - 8;

Memory[SS:RSP] ← SRC; (* push quadword *)

ELSE IF OperandSize = 32

THEN

RSP ← RSP - 4;

Memory[SS:RSP] ← SRC; (* push dword *)

ELSE (* OperandSize = 16 *)

RSP ← RSP - 2;

Memory[SS:RSP] ← SRC; (* push word *)

FI;

ELSE IF StackAddrSize = 32

THEN

IF OperandSize = 64

THEN

ESP ← ESP - 8;

Memory[SS:ESP] ← SRC; (* push quadword *)

ELSE IF OperandSize = 32

THEN

ESP ← ESP - 4;

Memory[SS:ESP] ← SRC; (* push dword *)

ELSE (* OperandSize = 16 *)

ESP ← ESP - 2;

Memory[SS:ESP] ← SRC; (* push word *)

FI;

ELSE (* StackAddrSize = 16 *)

```

    IF OperandSize = 32
    THEN
        SP ← SP - 4;
        Memory[SS:SP] ← SRC;          (* push dword *)
    ELSE (* OperandSize = 16 *)
        SP ← SP - 2;
        Memory[SS:SP] ← SRC;          (* push word *)
    FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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. If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used. If the PUSH is of CS, SS, DS, or ES.

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
60	PUSHA	NP	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	NP	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-bit Mode

THEN #UD

FI;

IF OperandSize = 32 (* PUSHAD instruction *)

THEN

Temp ← (ESP);
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temp);
Push(EBP);
Push(ESI);
Push(EDI);

ELSE (* OperandSize = 16, PUSHA instruction *)

Temp ← (SP);
Push(AX);
Push(CX);
Push(DX);

Push(BX);
 Push(Temp);
 Push(BP);
 Push(SI);
 Push(DI);

FI;

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.
-----	--------------------

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	NP	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	NP	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	NP	Valid	N.E.	Push RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFQ instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS register.

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

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

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

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Operation

IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))

(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)

THEN

IF OperandSize = 32

THEN

push (EFLAGS AND 00FCFFFFH);

(* VM and RF EFLAG bits are cleared in image stored on the stack *)

ELSE

push (EFLAGS); (* Lower 16 bits only *)

FI;

ELSE IF 64-bit MODE (* In 64-bit Mode *)

IF OperandSize = 64

```

    THEN
        push (RFLAGS AND 00000000_00FCFFFFH);
        (* VM and RF RFLAG bits are cleared in image stored on the stack; *)
    ELSE
        push (EFLAGS); (* Lower 16 bits only *)
FI;

ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
    #GP(0); (* Trap to virtual-8086 monitor *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EF /r ¹ PXOR <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Bitwise XOR of <i>mm/m64</i> and <i>mm</i> .
66 0F EF /r PXOR <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise XOR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG EF /r VPXOR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Bitwise XOR of <i>xmm3/m128</i> and <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG EF /r VPXOR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Bitwise XOR of <i>ymm3/m256</i> and <i>ymm2</i> .

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PXOR (128-bit Legacy SSE version)

DEST ← DEST XOR SRC

DEST[VLMAX-1:128] (Unmodified)

VPXOR (VEX.128 encoded version)

DEST ← SRC1 XOR SRC2

DEST[VLMAX-1:128] ← 0

VPXOR (VEX.256 encoded version)DEST \leftarrow SRC1 XOR SRC2**Intel C/C++ Compiler Intrinsic Equivalent**

PXOR: __m64 _mm_xor_si64 (__m64 m1, __m64 m2)

(V)PXOR: __m128i _mm_xor_si128 (__m128i a, __m128i b)

VPXOR: __m256i _mm256_xor_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

RCL/RCR/ROL/ROR—Rotate

Opcode**	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
D0 /2	RCL <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i>) left once.
REX + D0 /2	RCL <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i>) left once.
D2 /2	RCL <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i>) left CL times.
REX + D2 /2	RCL <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i>) left CL times.
C0 /2 <i>ib</i>	RCL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i>) left <i>imm8</i> times.
REX + C0 /2 <i>ib</i>	RCL <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i>) left <i>imm8</i> times.
D1 /2	RCL <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i>) left once.
D3 /2	RCL <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i>) left CL times.
C1 /2 <i>ib</i>	RCL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i>) left <i>imm8</i> times.
D1 /2	RCL <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i>) left once.
REX.W + D1 /2	RCL <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i>) left once. Uses a 6 bit count.
D3 /2	RCL <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i>) left CL times.
REX.W + D3 /2	RCL <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i>) left CL times. Uses a 6 bit count.
C1 /2 <i>ib</i>	RCL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i>) left <i>imm8</i> times.
REX.W + C1 /2 <i>ib</i>	RCL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i>) left <i>imm8</i> times. Uses a 6 bit count.
D0 /3	RCR <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i>) right once.
REX + D0 /3	RCR <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i>) right once.
D2 /3	RCR <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i>) right CL times.
REX + D2 /3	RCR <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i>) right CL times.
C0 /3 <i>ib</i>	RCR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i>) right <i>imm8</i> times.
REX + C0 /3 <i>ib</i>	RCR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i>) right <i>imm8</i> times.
D1 /3	RCR <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i>) right once.
D3 /3	RCR <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i>) right CL times.
C1 /3 <i>ib</i>	RCR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i>) right <i>imm8</i> times.
D1 /3	RCR <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i>) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i>) right once. Uses a 6 bit count.
D3 /3	RCR <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i>) right CL times.
REX.W + D3 /3	RCR <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i>) right CL times. Uses a 6 bit count.
C1 /3 <i>ib</i>	RCR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i>) right <i>imm8</i> times.
REX.W + C1 /3 <i>ib</i>	RCR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i>) right <i>imm8</i> times. Uses a 6 bit count.
D0 /0	ROL <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 8 bits <i>r/m8</i> left once.
REX + D0 /0	ROL <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left once
D2 /0	ROL <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 8 bits <i>r/m8</i> left CL times.
REX + D2 /0	ROL <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left CL times.
C0 /0 <i>ib</i>	ROL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + C0 /0 <i>ib</i>	ROL <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 16 bits <i>r/m16</i> left once.
D3 /0	ROL <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 16 bits <i>r/m16</i> left CL times.
C1 /0 <i>ib</i>	ROL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 16 bits <i>r/m16</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 32 bits <i>r/m32</i> left once.
REX.W + D1 /0	ROL <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left once. Uses a 6 bit count.
D3 /0	ROL <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 32 bits <i>r/m32</i> left CL times.
REX.W + D3 /0	ROL <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left CL times. Uses a 6 bit count.
C1 /0 <i>ib</i>	ROL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 32 bits <i>r/m32</i> left <i>imm8</i> times.
REX.W + C1 /0 <i>ib</i>	ROL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left <i>imm8</i> times. Uses a 6 bit count.
D0 /1	ROR <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 8 bits <i>r/m8</i> right once.
REX + D0 /1	ROR <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right once.
D2 /1	ROR <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 8 bits <i>r/m8</i> right CL times.
REX + D2 /1	ROR <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right CL times.
C0 /1 <i>ib</i>	ROR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
REX + C0 /1 <i>ib</i>	ROR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 16 bits <i>r/m16</i> right once.
D3 /1	ROR <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 16 bits <i>r/m16</i> right CL times.
C1 /1 <i>ib</i>	ROR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 16 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 32 bits <i>r/m32</i> right once.
REX.W + D1 /1	ROR <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right once. Uses a 6 bit count.
D3 /1	ROR <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 32 bits <i>r/m32</i> right CL times.
REX.W + D3 /1	ROR <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right CL times. Uses a 6 bit count.
C1 /1 <i>ib</i>	ROR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 32 bits <i>r/m32</i> right <i>imm8</i> times.
REX.W + C1 /1 <i>ib</i>	ROR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right <i>imm8</i> times. Uses a 6 bit count.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	NA	NA
MC	ModRM:r/m (w)	CL	NA	NA
MI	ModRM:r/m (w)	<i>imm8</i>	NA	NA

Description

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

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

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

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

(* RCL and RCR instructions *)

SIZE ← OperandSize;

CASE (determine count) OF

SIZE ← 8: tempCOUNT ← (COUNT AND 1FH) MOD 9;
 SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 17;
 SIZE ← 32: tempCOUNT ← COUNT AND 1FH;
 SIZE ← 64: tempCOUNT ← COUNT AND 3FH;

ESAC;

(* RCL instruction operation *)

WHILE (tempCOUNT ≠ 0)

DO

tempCF ← MSB(DEST);
 DEST ← (DEST * 2) + CF;
 CF ← tempCF;
 tempCOUNT ← tempCOUNT - 1;

OD;

ELIHW;

IF COUNT = 1

THEN OF ← MSB(DEST) XOR CF;
 ELSE OF is undefined;

FI;

(* RCR instruction operation *)

IF COUNT = 1

```

    THEN OF  $\leftarrow$  MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT  $\neq$  0)
    DO
        tempCF  $\leftarrow$  LSB(SRC);
        DEST  $\leftarrow$  (DEST / 2) + (CF * 2SIZE);
        CF  $\leftarrow$  tempCF;
        tempCOUNT  $\leftarrow$  tempCOUNT - 1;
    OD;

(* ROL and ROR instructions *)
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
tempCOUNT  $\leftarrow$  (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT  $\neq$  0)
    DO
        tempCF  $\leftarrow$  MSB(DEST);
        DEST  $\leftarrow$  (DEST * 2) + tempCF;
        tempCOUNT  $\leftarrow$  tempCOUNT - 1;
    OD;
ELIHW;
CF  $\leftarrow$  LSB(DEST);
IF (COUNT & COUNTMASK) = 1
    THEN OF  $\leftarrow$  MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* ROR instruction operation *)
tempCOUNT  $\leftarrow$  (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT  $\neq$  0)
    DO
        tempCF  $\leftarrow$  LSB(SRC);
        DEST  $\leftarrow$  (DEST / 2) + (tempCF * 2SIZE);
        tempCOUNT  $\leftarrow$  tempCOUNT - 1;
    OD;
ELIHW;
CF  $\leftarrow$  MSB(DEST);
IF (COUNT & COUNTMASK) = 1
    THEN OF  $\leftarrow$  MSB(DEST) XOR MSB - 1(DEST);
    ELSE OF is undefined;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment. If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 53 /r RCPPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Computes the approximate reciprocals of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.OF.WIG 53 /r VRCPPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in <i>xmm2/mem</i> and stores the results in <i>xmm1</i> .
VEX.256.OF.WIG 53 /r VRCPPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in <i>ymm2/mem</i> and stores the results in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.1111111110100000000000B * 2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B * 2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RCPPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[127:96])$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VRCPPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[127:96])$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VRCPPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[127:96])$
 $\text{DEST}[159:128] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[159:128])$
 $\text{DEST}[191:160] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[191:160])$
 $\text{DEST}[223:192] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[223:192])$
 $\text{DEST}[255:224] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[255:224])$

Intel C/C++ Compiler Intrinsic Equivalent

RCCPS: `__m128 _mm_rcp_ps(__m128 a)`
 RCPPS: `__m256 _mm256_rcp_ps (__m256 a);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv \neq 1111B.

RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 53 /r RCPSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm2/m32</i> and stores the result in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 53 /r VRCPPSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm3/m32</i> and stores the result in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.1111111110100000000000B * 2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B * 2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

RCPSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[VLMAX-1:32] (Unmodified)

VRCPPSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])

$\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS: `__m128 _mm_rcp_ss(__m128 a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

RDFSBASE/RDGSBASE—Read FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 OF AE /0 RDFSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the 32-bit destination register with the FS base address.
REX.W + F3 OF AE /0 RDFSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the 64-bit destination register with the FS base address.
F3 OF AE /1 RDGSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the 32-bit destination register with the GS base address.
REX.W + F3 OF AE /1 RDGSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the 64-bit destination register with the GS base address.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Loads the general-purpose register indicated by the modR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

Operation

DEST ← FS/GS segment base address;

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

```
RDFSBASE:    unsigned int _readfsbase_u32(void);
RDFSBASE:    unsigned __int64 _readfsbase_u64(void);
RDGSBASE:    unsigned int _readgsbase_u32(void);
RDGSBASE:    unsigned __int64 _readgsbase_u64(void);
```

Protected Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
 If CR4.FSGSBASE[bit 16] = 0.
 If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0.

RDMSR—Read from Model Specific Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 32	RDMSR	NP	Valid	Valid	Read MSR specified by ECX into EDX:EAX.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

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

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 35, “Model-Specific Registers (MSRs),” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

EDX:EAX ← MSR[ECX];

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP If the value in ECX specifies a reserved or unimplemented MSR address.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) The RDMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
OF 01 EE	RDPKRU	NP	V/V	OSPKE	Reads PKRU into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The RDPKRU instruction loads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, a general-protection exception (#GP) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

In 64-bit mode, bits 63:32 of RCX are ignored, and RDPKRU clears bits 63:32 of each of RDX and RAX.

Operation

```
IF (ECX = 0)
    THEN
        EAX ← PKRU;
        EDX ← 0;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0
#UD	If the LOCK prefix is used. If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 33	RDPMC	NP	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do not support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for “fast” reads.

The ECX register selects one of two type of performance counters, specifies the index relative to the base of each counter type, and selects “fast” read mode if supported. The two counter types are :

- General-purpose or special-purpose performance counters: The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Section 30.13, 30.14 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. This counter type is selected if ECX[30] is clear.
- Fixed-function performance counter. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

ECX[29:0] specifies the index. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific. The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf.

Table 4-13 lists valid indices of the general-purpose and special-purpose performance counters according to the derived DisplayFamily_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-M” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

Table 4-13. Valid General and Special Purpose Performance Counter Index Range for RDPMC

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH	0, 1	0, 1
Pentium® 4, Intel® Xeon processors	0FH_00H, 0FH_01H, 0FH_02H (0FH_03H, 0FH_04H, 0FH_06H) and (L3 is absent)	≥ 0 and ≤ 17 ≥ 0 and ≤ 17	≥ 0 and ≤ 17 ≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1

Table 4-13. Valid General and Special Purpose Performance Counter Index Range for RDPMC (Contd.)

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel Xeon processors 7100 series with L3	(0FH_06H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™2 Duo processor family, Intel Xeon processor family - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
Intel® Atom™ processor family	06H_1CH	0, 1	0, 1
Intel® Core™i7 processor, Intel Xeon processors 5500 series	06H_1AH, 06H_1EH, 06H_1FH, 06H_2EH	0-3	0, 1, 2, 3

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters. On Intel Xeon processor 7100 series with L3, performance counters with indices 18-25 are also 32-bit counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDPMC with ECX between from 4000_0000H and 4000_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, “Performance Monitoring Events,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

Operation

(* Intel Core i7 processor family and Intel Xeon processor 3400, 5500 series*)

Most significant counter bit (MSCB) = 47

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
  ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
```

FI;

(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series*)

Most significant counter bit (MSCB) = 39

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
  ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
```

FI;

(* P6 family processors and Pentium processor with MMX technology *)

```
IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN
    EAX ← PMC(ECX)[31:0];
    EDX ← PMC(ECX)[39:32];
  ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
```

FI;

(* Processors with CPUID family 15 *)

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN IF (ECX[30:0] = 0:17)
    THEN IF ECX[31] = 0
      THEN
        EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
        EDX ← PMC(ECX[30:0])[39:32];
      ELSE (* ECX[31] = 1 *)
        THEN
          EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
          EDX ← 0;
```

FI;

ELSE IF (*64-bit Intel Xeon processor with L3 *)

THEN IF (ECX[30:0] = 18:25)

```

    EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
    EDX ← 0;
FI;
ELSE IF (*Intel Xeon processor 7100 series with L3 *)
    THEN IF (ECX[30:0] = 18:25 )
        EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
        EDX ← 0;
    FI;
ELSE (* Invalid PMC index in ECX[30:0], see Table 4-16. *)
    GP(0);
FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
 If an invalid performance counter index is specified (see Table 4-13).
 (Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If an invalid performance counter index is specified (see Table 4-13).
 (Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the PCE flag in the CR4 register is clear.
 If an invalid performance counter index is specified (see Table 4-13).
 (Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
 If an invalid performance counter index is specified in ECX[30:0] (see Table 4-13).

#UD If the LOCK prefix is used.

RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
OF C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
REX.W + OF C7 /6 RDRAND r64	M	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 7.3.17, "Random Number Generator Instructions").

This instruction is available at all privilege levels.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF HW_RND_GEN.ready = 1
  THEN
    CASE of
      osize is 64: DEST[63:0] ← HW_RND_GEN.data;
      osize is 32: DEST[31:0] ← HW_RND_GEN.data;
      osize is 16: DEST[15:0] ← HW_RND_GEN.data;
    ESAC
    CF ← 1;
  ELSE
    CASE of
      osize is 64: DEST[63:0] ← 0;
      osize is 32: DEST[31:0] ← 0;
      osize is 16: DEST[15:0] ← 0;
    ESAC
    CF ← 0;
  FI
OF, SF, ZF, AF, PF ← 0;

```

Flags Affected

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

Intel C/C++ Compiler Intrinsic Equivalent

RDRAND: `int _rdrand16_step(unsigned short *);`
 RDRAND: `int _rdrand32_step(unsigned int *);`
 RDRAND: `int _rdrand64_step(unsigned __int64 *);`

Protected Mode Exceptions

#UD If the LOCK prefix is used.
 If the F2H or F3H prefix is used.
 If CPUID.01H:ECX.RDRAND[bit 30] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C7 /7 RDSEED r16	M	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
OF C7 /7 RDSEED r32	M	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
REX.W + OF C7 /7 RDSEED r64	M	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF HW_NRND_GEN.ready = 1

THEN

CASE of

osize is 64: DEST[63:0] ← HW_NRND_GEN.data;

osize is 32: DEST[31:0] ← HW_NRND_GEN.data;

osize is 16: DEST[15:0] ← HW_NRND_GEN.data;

ESAC;

CF ← 1;

ELSE

CASE of

osize is 64: DEST[63:0] ← 0;

osize is 32: DEST[31:0] ← 0;

osize is 16: DEST[15:0] ← 0;

ESAC;

CF ← 0;

FI;

OF, SF, ZF, AF, PF ← 0;

Flags Affected

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

C/C++ Compiler Intrinsic Equivalent

```
RDSEED int _rdseed16_step( unsigned short * );
RDSEED int _rdseed32_step( unsigned int * );
RDSEED int _rdseed64_step( unsigned __int64 * );
```

Protected Mode Exceptions

#UD If the LOCK prefix is used.
 If the F2H or F3H prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
 If the F2H or F3H prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
 If the F2H or F3H prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
 If the F2H or F3H prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
 If the F2H or F3H prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 31	RDTSC	NP	Valid	Valid	Read time-stamp counter into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. If software requires RDTSC to be executed only after all previous instructions have completed locally, it can either use RDTSCP (if the processor supports that instruction) or execute the sequence LFENCE;RDTSC.

This instruction was introduced by the Pentium processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 F9	RDTSCP	NP	Valid	Valid	Read 64-bit time-stamp counter and 32-bit IA32_TSC_AUX value into EDX:EAX and ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also loads the IA32_TSC_AUX MSR (address C000_0103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.

The presence of the RDTSCP instruction is indicated by CPUID leaf 80000001H, EDX bit 27. If the bit is set to 1 then RDTSCP is present on the processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX ← TimeStampCounter;
        ECX ← IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
 #UD If the LOCK prefix is used.
 If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

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

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS <i>m8</i> , DX	NP	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS <i>m8</i> , DX	NP	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS <i>m16</i> , DX	NP	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI].
F3 6D	REP INS <i>m32</i> , DX	NP	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS <i>r/m32</i> , DX	NP	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS <i>m8</i> , <i>m8</i>	NP	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS <i>m8</i> , <i>m8</i>	NP	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS <i>m16</i> , <i>m16</i>	NP	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS <i>m32</i> , <i>m32</i>	NP	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS <i>m64</i> , <i>m64</i>	NP	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, <i>r/m8</i>	NP	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, <i>r/m8</i> *	NP	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, <i>r/m16</i>	NP	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, <i>r/m32</i>	NP	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, <i>r/m32</i>	NP	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	NP	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS AL	NP	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	NP	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	NP	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS RAX	NP	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS <i>m8</i>	NP	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS <i>m8</i>	NP	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS <i>m16</i>	NP	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS <i>m32</i>	NP	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS <i>m64</i>	NP	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS <i>m8</i> , <i>m8</i>	NP	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS <i>m8</i> , <i>m8</i>	NP	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS <i>m16</i> , <i>m16</i>	NP	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS <i>m32</i> , <i>m32</i>	NP	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS <i>m64</i> , <i>m64</i>	NP	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS <i>m8</i>	NP	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS <i>m8</i>	NP	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS <i>m16</i>	NP	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS <i>m32</i>	NP	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 REX.W AF	REPE SCAS <i>m64</i>	NP	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS <i>m8, m8</i>	NP	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS <i>m8, m8</i>	NP	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS <i>m16, m16</i>	NP	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS <i>m32, m32</i>	NP	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS <i>m64, m64</i>	NP	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS <i>m8</i>	NP	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS <i>m8</i>	NP	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS <i>m16</i>	NP	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS <i>m32</i>	NP	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS <i>m64</i>	NP	Valid	N.E.	Find RAX, starting at [RDI].

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The F3H prefix is defined for the following instructions and undefined for the rest:

- F3H as REP/REPE/REPZ for string and input/output instruction.
- F3H is a mandatory prefix for POPCNT, LZCNT, and ADOX.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-14.

Table 4-14. Repeat Prefixes

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

NOTES:

* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

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

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

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

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
        Implicit Source/Dest operand for memory use of SI/DI;
    ELSE IF AddressSize = 64
        THEN Use RCX for CountReg;
        Implicit Source/Dest operand for memory use of RSI/RDI;
    ELSE
        Use ECX for CountReg;
        Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg - 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
        or (Repeat prefix is REPZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
    OD;
```

Flags Affected

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

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.

RET—Return from Procedure

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	NP	Valid	Valid	Near return to calling procedure.
CB	RET	NP	Valid	Valid	Far return to calling procedure.
C2 <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Near return to calling procedure and pop <i>imm16</i> bytes from stack.
CA <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Far return to calling procedure and pop <i>imm16</i> bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	<i>imm16</i>	NA	NA	NA

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

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

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

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

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

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

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

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

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

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

Operation

(* Near return *)

IF instruction = near return

THEN;

IF OperandSize = 32

THEN

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP \leftarrow Pop();

ELSE

IF OperandSize = 64

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

RIP \leftarrow Pop();

ELSE (* OperandSize = 16 *)

IF top 2 bytes of stack not within stack limits

THEN #SS(0); FI;

tempEIP \leftarrow Pop();

tempEIP \leftarrow tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits

THEN #GP(0); FI;

EIP \leftarrow tempEIP;

FI;

FI;

IF instruction has immediate operand

THEN (* Release parameters from stack *)

IF StackAddressSize = 32

THEN

ESP \leftarrow ESP + SRC;

ELSE

IF StackAddressSize = 64

THEN

RSP \leftarrow RSP + SRC;

ELSE (* StackAddressSize = 16 *)

SP \leftarrow SP + SRC;

FI;

FI;

FI;

FI;

(* Real-address mode or virtual-8086 mode *)

IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return

THEN

IF OperandSize = 32

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP \leftarrow Pop();

CS \leftarrow Pop(); (* 32-bit pop, high-order 16 bits discarded *)

ELSE (* OperandSize = 16 *)

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

```

        tempEIP ← Pop();
        tempEIP ← tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits
            THEN #GP(0); FI;
        EIP ← tempEIP;
        CS ← Pop(); (* 16-bit pop *)
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP ← SP + (SRC AND FFFFH);
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                ELSE (* OperandSize = 16 *)
                    IF second word on stack is not within stack limits
                        THEN #SS(0); FI;
            FI;
        IF return code segment selector is NULL
            THEN #GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN #GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment
            THEN #GP(selector); FI;
        IF return code segment selector RPL < CPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is conforming
        and return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is non-conforming and return code
        segment DPL ≠ return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is not present
            THEN #NP(selector); FI;
        IF return code segment selector RPL > CPL
            THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

```

RETURN-SAME-PRIVILEGE-LEVEL:

```

    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    FI;

```

```

    ELSE (* OperandSize = 16 *)
        EIP ← Pop();
        EIP ← EIP AND 0000FFFFH;
        CS ← Pop(); (* 16-bit pop *)
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
        FI;
FI;

RETURN-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
        THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
        THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();

```

```

EIP ← EIP AND 0000FFFFH;
CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
CS(RPL) ← CPL;
IF instruction has immediate operand
    THEN (* Release parameters from called procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
        FI;
    FI;
tempESP ← Pop();
tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
ESP ← tempESP;
SS ← tempSS;
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
            THEN SegmentSelector ← 0; (* Segment selector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
        FI;
    FI;

(* IA-32e Mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                IF first or second doubleword on stack is not in canonical space
                    THEN #SS(0); FI;
            ELSE
                IF OperandSize = 16
                    THEN
                        IF second word on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second word on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE (* OperandSize = 64 *)
                        IF first or second quadword on stack is not in canonical space

```

```

        THEN #SS(0); FI;
    FI;
    IF return code segment selector is NULL
        THEN GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN GP(selector); FI;
    IF return code segment selector addresses descriptor in non-canonical space
        THEN GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment descriptor has L-bit = 1 and D-bit = 1
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming
    and return code segment DPL ≠ return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
    FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN

```

```

        ESP ← ESP + SRC;
    ELSE
        IF StackAddressSize = 16
            THEN
                SP ← SP + SRC;
            ELSE (* StackAddressSize = 64 *)
                RSP ← RSP + SRC;
        FI;
    FI;
FI;

```

IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL:

```

IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
    FI;
IF return stack segment descriptor is not within descriptor table limits
    THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE

```

```

        IF StackAddressSize = 16
            THEN
                SP  $\leftarrow$  SP + SRC;
            ELSE (* StackAddressSize = 64 *)
                RSP  $\leftarrow$  RSP + SRC;
        FI;
    FI;
FI;
tempESP  $\leftarrow$  Pop();
tempSS  $\leftarrow$  Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
ESP  $\leftarrow$  tempESP;
SS  $\leftarrow$  tempSS;
ELSE
    IF OperandSize = 16
        THEN
            EIP  $\leftarrow$  Pop();
            EIP  $\leftarrow$  EIP AND 0000FFFFH;
            CS  $\leftarrow$  Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL)  $\leftarrow$  CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP  $\leftarrow$  ESP + SRC;
                        ELSE
                            IF StackAddressSize = 16
                                THEN
                                    SP  $\leftarrow$  SP + SRC;
                                ELSE (* StackAddressSize = 64 *)
                                    RSP  $\leftarrow$  RSP + SRC;
                            FI;
                        FI;
                    FI;
                tempESP  $\leftarrow$  Pop();
                tempSS  $\leftarrow$  Pop(); (* 16-bit pop; segment descriptor loaded *)
                ESP  $\leftarrow$  tempESP;
                SS  $\leftarrow$  tempSS;
            ELSE (* OperandSize = 64 *)
                RIP  $\leftarrow$  Pop();
                CS  $\leftarrow$  Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                CS(RPL)  $\leftarrow$  CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        RSP  $\leftarrow$  RSP + SRC;
                    FI;
                tempESP  $\leftarrow$  Pop();
                tempSS  $\leftarrow$  Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                ESP  $\leftarrow$  tempESP;
                SS  $\leftarrow$  tempSS;
            FI;
        FI;
FI;

```

FOR each of segment register (ES, FS, GS, and DS)

DO


```

    IF segment register points to data or non-conforming code segment
    and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
    THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
    FI;
OD;

IF instruction has immediate operand
THEN (* Release parameters from calling procedure's stack *)
    IF StackAddressSize = 32
    THEN
        ESP ← ESP + SRC;
    ELSE
        IF StackAddressSize = 16
        THEN
            SP ← SP + SRC;
        ELSE (* StackAddressSize = 64 *)
            RSP ← RSP + SRC;
        FI;
    FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector NULL.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less than the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
--------	---

#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	<p>If the return instruction pointer is non-canonical.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If the stack segment selector is NULL going back to compatibility mode.</p> <p>If the stack segment selector is NULL going back to CPL3 64-bit mode.</p> <p>If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.</p> <p>If the return code segment selector is NULL.</p>
#GP(selector)	<p>If the proposed segment descriptor for a code segment does not indicate it is a code segment.</p> <p>If the proposed new code segment descriptor has both the D-bit and L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p> <p>If CPL is greater than the RPL of the code segment selector.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If a segment selector index is outside its descriptor table limits.</p> <p>If a segment descriptor memory address is non-canonical.</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If an attempt to pop a value off the stack violates the SS limit.</p> <p>If an attempt to pop a value off the stack causes a non-canonical address to be referenced.</p>
#NP(selector)	If the return code or stack segment is not present.
#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.

RORX — Rotate Right Logical Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, r/m32, imm8	RMI	V/V	BMI2	Rotate 32-bit <i>r/m32</i> right <i>imm8</i> times without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, r/m64, imm8	RMI	V/N.E.	BMI2	Rotate 64-bit <i>r/m64</i> right <i>imm8</i> times without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
IF (OperandSize = 32)
    y ← imm8 AND 1FH;
    DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64 )
    y ← imm8 AND 3FH;
    DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally #UD If VEX.W = 1.

ROUNDPD — Round Packed Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 09 /r ib ROUNDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Round packed double precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Round packed double-precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX	Round packed double-precision floating-point values in <i>ymm2/m256</i> and place the result in <i>ymm1</i> . The rounding mode is determined by <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-15 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

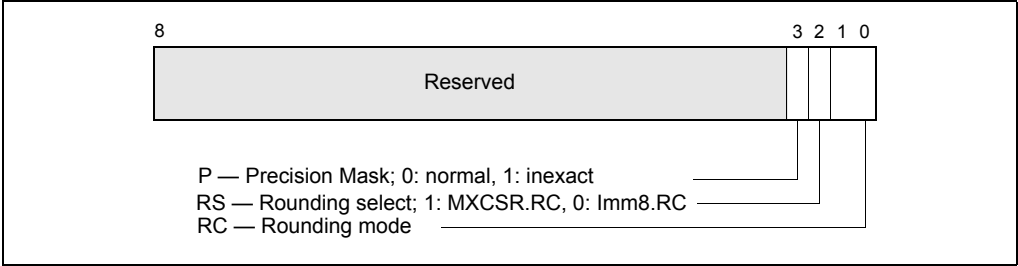


Figure 4-20. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-15. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward $-\infty$)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward $+\infty$)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Operation

```
IF (imm[2] = '1')
    THEN // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI
```

ROUNDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[VLMAX-1:128] (Unmodified)
```

VROUNDPD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[VLMAX-1:128] ← 0
```

VROUNDPD (VEX.256 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[191:128] ← RoundToInteger(SRC[191:128], ROUND_CONTROL)
DEST[255:192] ← RoundToInteger(SRC[255:192], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
__m128 _mm_round_pd(__m128d s1, int iRoundMode);
```

```

__m128 _mm_floor_pd(__m128d s1);
__m128 _mm_ceil_pd(__m128d s1)
__m256 _mm256_round_pd(__m256d s1, int iRoundMode);
__m256 _mm256_floor_pd(__m256d s1);
__m256 _mm256_ceil_pd(__m256d s1)

```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0'; if imm[3] = '1', then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

ROUNDPS — Round Packed Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 08 /r ib ROUNDPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Round packed single precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Round packed single-precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX	Round packed single-precision floating-point values in <i>ymm2/m256</i> and place the result in <i>ymm1</i> . The rounding mode is determined by <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-15 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```

IF (imm[2] = '1')
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;

```

ROUNDPS(128-bit Legacy SSE version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[VLMAX-1:128] (Unmodified)

```

VROUNDPS (VEX.128 encoded version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[VLMAX-1:128] ← 0

```

VROUNDPS (VEX.256 encoded version)

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[159:128] ← RoundToInteger(SRC[159:128], ROUND_CONTROL)
DEST[191:160] ← RoundToInteger(SRC[191:160], ROUND_CONTROL)
DEST[223:192] ← RoundToInteger(SRC[223:192], ROUND_CONTROL)
DEST[255:224] ← RoundToInteger(SRC[255:224], ROUND_CONTROL)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

__m128 _mm_round_ps(__m128 s1, int iRoundMode);
__m128 _mm_floor_ps(__m128 s1);
__m128 _mm_ceil_ps(__m128 s1)
__m256 _mm256_round_ps(__m256 s1, int iRoundMode);
__m256 _mm256_floor_ps(__m256 s1);
__m256 _mm256_ceil_ps(__m256 s1)

```


SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

ROUNDSD — Round Scalar Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0B /r ib ROUNDSD <i>xmm1</i> , <i>xmm2/m64</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Round the low packed double precision floating-point value in <i>xmm2/m64</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.NDS.LIG.66.0F3A.WIG 0B /r ib VROUNDSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m64</i> , <i>imm8</i>	RVMI	V/V	AVX	Round the low packed double precision floating-point value in <i>xmm3/m64</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> . Upper packed double precision floating-point value (bits[127:64]) from <i>xmm2</i> is copied to <i>xmm1</i> [127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-15 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
            DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
            DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

ROUNDSD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[VLMAX-1:64] (Unmodified)
```

VROUNDSD (VEX.128 encoded version)

DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

```
ROUNDSD:    __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
             __m128d mm_floor_sd(__m128d dst, __m128d s1);
             __m128d mm_ceil_sd(__m128d dst, __m128d s1);
```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0'; if imm[3] = '1', then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Exceptions Type 3.

ROUNDSS — Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A /r ib ROUNDSS <i>xmm1</i> , <i>xmm2/m32</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Round the low packed single precision floating-point value in <i>xmm2/m32</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.NDS.LIG.66.0F3A.WIG 0A /r ib VROUNDSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i> , <i>imm8</i>	RVMI	V/V	AVX	Round the low packed single precision floating-point value in <i>xmm3/m32</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> . Also, upper packed single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer value and returns the result as a single-precision floating-point value in the lowest position. The upper three single-precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-15 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1')
    THEN    // rounding mode is determined by MXCSR.RC
            DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE    // rounding mode is determined by IMM8.RC
            DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

ROUNDSS (128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[VLMAX-1:32] (Unmodified)
```

VROUNDSS (VEX.128 encoded version)

DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

```
ROUNDSS:    __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
             __m128 mm_floor_ss(__m128 dst, __m128 s1);
             __m128 mm_ceil_ss(__m128 dst, __m128 s1);
```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Exceptions Type 3.

RSM—Resume from System Management Mode

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AA	RSM	NP	Invalid	Valid	Resume operation of interrupted program.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

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

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 34, "System Management Mode," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about SMM and the behavior of the RSM instruction.

Operation

ReturnFromSMM;

IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH)

THEN

ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));

Else

ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));

FI

Flags Affected

All.

Protected Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.
If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 52 /r RSQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.OF.WIG 52 /r VRSQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in <i>xmm2/mem</i> and stores the results in <i>xmm1</i> .
VEX.256.OF.WIG 52 /r VRSQRTPS <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in <i>ymm2/mem</i> and stores the results in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RSQRTPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC}[31:0]))$
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[63:32]))$
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[95:64]))$
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[127:96]))$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VRSQRTPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC}[31:0]))$
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[63:32]))$
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[95:64]))$
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[127:96]))$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VRSQRTPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC}[31:0]))$
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[63:32]))$
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[95:64]))$
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[127:96]))$
 $\text{DEST}[159:128] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[159:128]))$
 $\text{DEST}[191:160] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[191:160]))$
 $\text{DEST}[223:192] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[223:192]))$
 $\text{DEST}[255:224] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[255:224]))$

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS: `__m128 _mm_rsqrt_ps(__m128 a)`
 RSQRTPS: `__m256 _mm256_rsqrt_ps (__m256 a);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv \neq 1111B.

RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 52 /r RSQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 52 /r VRSQRTSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in <i>xmm3/m32</i> and stores the results in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

RSQRTSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))

DEST[VLMAX-1:32] (Unmodified)

VRSQRTSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS: `__m128_mm_rsqrt_ss(__m128 a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

SAHF—Store AH into Flags

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9E	SAHF	NP	Invalid*	Valid	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.

NOTES:

* Valid in specific steppings. See Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation

```
IF IA-64 Mode
  THEN
    IF CPUID.80000001H:ECX[0] = 1;
      THEN
        RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
      ELSE
        #UD;
    FI
  ELSE
    EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
FI;
```

Flags Affected

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

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Compatibility Mode Exceptions

None.

64-Bit Mode Exceptions

#UD	If CPUID.80000001H.ECX[0] = 0. If the LOCK prefix is used.
-----	---

SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL <i>r/m8</i> , 1	M1	Valid	Valid	Multiply <i>r/m8</i> by 2, once.
REX + D0 /4	SAL <i>r/m8**</i> , 1	M1	Valid	N.E.	Multiply <i>r/m8</i> by 2, once.
D2 /4	SAL <i>r/m8</i> , CL	MC	Valid	Valid	Multiply <i>r/m8</i> by 2, CL times.
REX + D2 /4	SAL <i>r/m8**</i> , CL	MC	Valid	N.E.	Multiply <i>r/m8</i> by 2, CL times.
C0 /4 <i>ib</i>	SAL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /4 <i>ib</i>	SAL <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /4	SAL <i>r/m16</i> , 1	M1	Valid	Valid	Multiply <i>r/m16</i> by 2, once.
D3 /4	SAL <i>r/m16</i> , CL	MC	Valid	Valid	Multiply <i>r/m16</i> by 2, CL times.
C1 /4 <i>ib</i>	SAL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /4	SAL <i>r/m32</i> , 1	M1	Valid	Valid	Multiply <i>r/m32</i> by 2, once.
REX.W + D1 /4	SAL <i>r/m64</i> , 1	M1	Valid	N.E.	Multiply <i>r/m64</i> by 2, once.
D3 /4	SAL <i>r/m32</i> , CL	MC	Valid	Valid	Multiply <i>r/m32</i> by 2, CL times.
REX.W + D3 /4	SAL <i>r/m64</i> , CL	MC	Valid	N.E.	Multiply <i>r/m64</i> by 2, CL times.
C1 /4 <i>ib</i>	SAL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /4 <i>ib</i>	SAL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m64</i> by 2, <i>imm8</i> times.
D0 /7	SAR <i>r/m8</i> , 1	M1	Valid	Valid	Signed divide* <i>r/m8</i> by 2, once.
REX + D0 /7	SAR <i>r/m8**</i> , 1	M1	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, once.
D2 /7	SAR <i>r/m8</i> , CL	MC	Valid	Valid	Signed divide* <i>r/m8</i> by 2, CL times.
REX + D2 /7	SAR <i>r/m8**</i> , CL	MC	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, CL times.
C0 /7 <i>ib</i>	SAR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> time.
REX + C0 /7 <i>ib</i>	SAR <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /7	SAR <i>r/m16</i> , 1	M1	Valid	Valid	Signed divide* <i>r/m16</i> by 2, once.
D3 /7	SAR <i>r/m16</i> , CL	MC	Valid	Valid	Signed divide* <i>r/m16</i> by 2, CL times.
C1 /7 <i>ib</i>	SAR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Signed divide* <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /7	SAR <i>r/m32</i> , 1	M1	Valid	Valid	Signed divide* <i>r/m32</i> by 2, once.
REX.W + D1 /7	SAR <i>r/m64</i> , 1	M1	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, once.
D3 /7	SAR <i>r/m32</i> , CL	MC	Valid	Valid	Signed divide* <i>r/m32</i> by 2, CL times.
REX.W + D3 /7	SAR <i>r/m64</i> , CL	MC	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, CL times.
C1 /7 <i>ib</i>	SAR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Signed divide* <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /7 <i>ib</i>	SAR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, <i>imm8</i> times
D0 /4	SHL <i>r/m8</i> , 1	M1	Valid	Valid	Multiply <i>r/m8</i> by 2, once.
REX + D0 /4	SHL <i>r/m8**</i> , 1	M1	Valid	N.E.	Multiply <i>r/m8</i> by 2, once.
D2 /4	SHL <i>r/m8</i> , CL	MC	Valid	Valid	Multiply <i>r/m8</i> by 2, CL times.
REX + D2 /4	SHL <i>r/m8**</i> , CL	MC	Valid	N.E.	Multiply <i>r/m8</i> by 2, CL times.
C0 /4 <i>ib</i>	SHL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /4 <i>ib</i>	SHL <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /4	SHL <i>r/m16</i> , 1	M1	Valid	Valid	Multiply <i>r/m16</i> by 2, once.
D3 /4	SHL <i>r/m16</i> , CL	MC	Valid	Valid	Multiply <i>r/m16</i> by 2, CL times.
C1 /4 <i>ib</i>	SHL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /4	SHL <i>r/m32</i> , 1	M1	Valid	Valid	Multiply <i>r/m32</i> by 2, once.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D1 /4	SHL <i>r/m64</i> ,1	M1	Valid	N.E.	Multiply <i>r/m64</i> by 2, once.
D3 /4	SHL <i>r/m32</i> , CL	MC	Valid	Valid	Multiply <i>r/m32</i> by 2, CL times.
REX.W + D3 /4	SHL <i>r/m64</i> , CL	MC	Valid	N.E.	Multiply <i>r/m64</i> by 2, CL times.
C1 /4 <i>ib</i>	SHL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /4 <i>ib</i>	SHL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m64</i> by 2, <i>imm8</i> times.
D0 /5	SHR <i>r/m8</i> ,1	M1	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, once.
REX + D0 /5	SHR <i>r/m8**</i> , 1	M1	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, once.
D2 /5	SHR <i>r/m8</i> , CL	MC	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, CL times.
REX + D2 /5	SHR <i>r/m8**</i> , CL	MC	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, CL times.
C0 /5 <i>ib</i>	SHR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /5 <i>ib</i>	SHR <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /5	SHR <i>r/m16</i> , 1	M1	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, once.
D3 /5	SHR <i>r/m16</i> , CL	MC	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, CL times
C1 /5 <i>ib</i>	SHR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /5	SHR <i>r/m32</i> , 1	M1	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, once.
REX.W + D1 /5	SHR <i>r/m64</i> , 1	M1	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, once.
D3 /5	SHR <i>r/m32</i> , CL	MC	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, CL times.
REX.W + D3 /5	SHR <i>r/m64</i> , CL	MC	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, CL times.
C1 /5 <i>ib</i>	SHR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /5 <i>ib</i>	SHR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, <i>imm8</i> times.

NOTES:

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

** In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

***See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (<i>r</i> , <i>w</i>)	1	NA	NA
MC	ModRM:r/m (<i>r</i> , <i>w</i>)	CL	NA	NA
MI	ModRM:r/m (<i>r</i> , <i>w</i>)	<i>imm8</i>	NA	NA

Description

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

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

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

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

In 64-bit mode, the instruction's default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

IF 64-Bit Mode and using REX.W

THEN

countMASK \leftarrow 3FH;

ELSE

countMASK \leftarrow 1FH;

FI

tempCOUNT \leftarrow (COUNT AND countMASK);

tempDEST \leftarrow DEST;

WHILE (tempCOUNT \neq 0)

DO

IF instruction is SAL or SHL

THEN

CF \leftarrow MSB(DEST);

ELSE (* Instruction is SAR or SHR *)

CF \leftarrow LSB(DEST);

FI;

IF instruction is SAL or SHL

THEN

DEST \leftarrow DEST * 2;

ELSE

IF instruction is SAR


```

        THEN
            DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
        ELSE (* Instruction is SHR *)
            DEST ← DEST / 2; (* Unsigned divide *)
    FI;
FI;
tempCOUNT ← tempCOUNT - 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
            FI;
        ELSE IF (COUNT AND countMASK) = 0
            THEN
                All flags unchanged;
            ELSE (* COUNT not 1 or 0 *)
                OF ← undefined;
        FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

SARX/SHLX/SHRX – Shift Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS ¹ .LZ.F3.0F38.W0 F7 /r SARX <i>r32a</i> , <i>r/m32</i> , <i>r32b</i>	RMV	V/V	BMI2	Shift <i>r/m32</i> arithmetically right with count specified in <i>r32b</i> .
VEX.NDS ¹ .LZ.66.0F38.W0 F7 /r SHLX <i>r32a</i> , <i>r/m32</i> , <i>r32b</i>	RMV	V/V	BMI2	Shift <i>r/m32</i> logically left with count specified in <i>r32b</i> .
VEX.NDS ¹ .LZ.F2.0F38.W0 F7 /r SHRX <i>r32a</i> , <i>r/m32</i> , <i>r32b</i>	RMV	V/V	BMI2	Shift <i>r/m32</i> logically right with count specified in <i>r32b</i> .
VEX.NDS ¹ .LZ.F3.0F38.W1 F7 /r SARX <i>r64a</i> , <i>r/m64</i> , <i>r64b</i>	RMV	V/N.E.	BMI2	Shift <i>r/m64</i> arithmetically right with count specified in <i>r64b</i> .
VEX.NDS ¹ .LZ.66.0F38.W1 F7 /r SHLX <i>r64a</i> , <i>r/m64</i> , <i>r64b</i>	RMV	V/N.E.	BMI2	Shift <i>r/m64</i> logically left with count specified in <i>r64b</i> .
VEX.NDS ¹ .LZ.F2.0F38.W1 F7 /r SHRX <i>r64a</i> , <i>r/m64</i> , <i>r64b</i>	RMV	V/N.E.	BMI2	Shift <i>r/m64</i> logically right with count specified in <i>r64b</i> .

NOTES:

1. ModRM:r/m is used to encode the first source operand (second operand) and VEX.vvvv encodes the second source operand (third operand).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize -1, the COUNT value is masked.

SARX,SHRX, and SHLX instructions do not update flags.

Operation

```
TEMP ← SRC1;
IF VEX.W1 and CS.L = 1
THEN
    countMASK ← 3FH;
```

```

ELSE
    countMASK  $\leftarrow$  1FH;
FI
COUNT  $\leftarrow$  (SRC2 AND countMASK)

DEST[OperandSize - 1] = TEMP[OperandSize - 1];
DO WHILE (COUNT  $\neq$  0)
    IF instruction is SHLX
        THEN
            DEST[]  $\leftarrow$  DEST * 2;
        ELSE IF instruction is SHRXL
            THEN
                DEST[]  $\leftarrow$  DEST / 2; //unsigned divide
            ELSE
                // SARXL
                DEST[]  $\leftarrow$  DEST / 2; // signed divide, round toward negative infinity
        FI;
    COUNT  $\leftarrow$  COUNT - 1;
OD

```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
1C <i>ib</i>	SBB AL, <i>imm8</i>	I	Valid	Valid	Subtract with borrow <i>imm8</i> from AL.
1D <i>iw</i>	SBB AX, <i>imm16</i>	I	Valid	Valid	Subtract with borrow <i>imm16</i> from AX.
1D <i>id</i>	SBB EAX, <i>imm32</i>	I	Valid	Valid	Subtract with borrow <i>imm32</i> from EAX.
REX.W + 1D <i>id</i>	SBB RAX, <i>imm32</i>	I	Valid	N.E.	Subtract with borrow sign-extended <i>imm.32</i> to 64-bits from RAX.
80 /3 <i>ib</i>	SBB <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Subtract with borrow <i>imm8</i> from <i>r/m8</i> .
REX + 80 /3 <i>ib</i>	SBB <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract with borrow <i>imm8</i> from <i>r/m8</i> .
81 /3 <i>iw</i>	SBB <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Subtract with borrow <i>imm16</i> from <i>r/m16</i> .
81 /3 <i>id</i>	SBB <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Subtract with borrow <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /3 <i>id</i>	SBB <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from <i>r/m64</i> .
83 /3 <i>ib</i>	SBB <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /3 <i>ib</i>	SBB <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /3 <i>ib</i>	SBB <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m64</i> .
18 /r	SBB <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Subtract with borrow <i>r8</i> from <i>r/m8</i> .
REX + 18 /r	SBB <i>r/m8*</i> , <i>r8</i>	MR	Valid	N.E.	Subtract with borrow <i>r8</i> from <i>r/m8</i> .
19 /r	SBB <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Subtract with borrow <i>r16</i> from <i>r/m16</i> .
19 /r	SBB <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Subtract with borrow <i>r32</i> from <i>r/m32</i> .
REX.W + 19 /r	SBB <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Subtract with borrow <i>r64</i> from <i>r/m64</i> .
1A /r	SBB <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Subtract with borrow <i>r/m8</i> from <i>r8</i> .
REX + 1A /r	SBB <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Subtract with borrow <i>r/m8</i> from <i>r8</i> .
1B /r	SBB <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Subtract with borrow <i>r/m16</i> from <i>r16</i> .
1B /r	SBB <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Subtract with borrow <i>r/m32</i> from <i>r32</i> .
REX.W + 1B /r	SBB <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Subtract with borrow <i>r/m64</i> from <i>r64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (w)	imm8/16/32	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

$DEST \leftarrow (DEST - (SRC + CF));$

Intel C/C++ Compiler Intrinsic Equivalent

SBB: extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);

SBB: extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);

SBB: extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *diff_out);

SBB: extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
--------	---

#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AE	SCAS <i>m8</i>	NP	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.*
AF	SCAS <i>m16</i>	NP	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.*
AF	SCAS <i>m32</i>	NP	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCAS <i>m64</i>	NP	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	NP	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.*
AF	SCASW	NP	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.*
AF	SCASD	NP	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCASQ	NP	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

NOTES:

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

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

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

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

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes

some action based on the setting of status flags. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

Operation

Non-64-bit Mode:

```
IF (Byte comparison)
  THEN
    temp ← AL – SRC;
    SetStatusFlags(temp);
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI – 1; FI;
  ELSE IF (Word comparison)
    THEN
      temp ← AX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (E)DI ← (E)DI + 2;
        ELSE (E)DI ← (E)DI – 2; FI;
      FI;
  ELSE IF (Doubleword comparison)
    THEN
      temp ← EAX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI – 4; FI;
      FI;
  FI;
```

64-bit Mode:

```
IF (Byte comparison)
  THEN
    temp ← AL – SRC;
    SetStatusFlags(temp);
    THEN IF DF = 0
      THEN (R)E)DI ← (R)E)DI + 1;
      ELSE (R)E)DI ← (R)E)DI – 1; FI;
  ELSE IF (Word comparison)
    THEN
      temp ← AX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (R)E)DI ← (R)E)DI + 2;
        ELSE (R)E)DI ← (R)E)DI – 2; FI;
      FI;
```

```

ELSE IF (Doubleword comparison)
    THEN
        temp ← EAX - SRC;
        SetStatusFlags(temp);
        IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 4;
            ELSE (R|E)DI ← (R|E)DI - 4; FI;
    FI;
ELSE IF (Quadword comparison using REX.W )
    THEN
        temp ← RAX - SRC;
        SetStatusFlags(temp);
        IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 8;
            ELSE (R|E)DI ← (R|E)DI - 8;
        FI;
    FI;
F

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a NULL segment selector. If an illegal memory operand effective address in the ES segment is given.
#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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

SETcc—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA <i>r/m8</i>	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 97	SETA <i>r/m8</i> *	M	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE <i>r/m8</i>	M	Valid	Valid	Set byte if above or equal (CF=0).
REX + 0F 93	SETAE <i>r/m8</i> *	M	Valid	N.E.	Set byte if above or equal (CF=0).
0F 92	SETB <i>r/m8</i>	M	Valid	Valid	Set byte if below (CF=1).
REX + 0F 92	SETB <i>r/m8</i> *	M	Valid	N.E.	Set byte if below (CF=1).
0F 96	SETBE <i>r/m8</i>	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + 0F 96	SETBE <i>r/m8</i> *	M	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC <i>r/m8</i>	M	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC <i>r/m8</i> *	M	Valid	N.E.	Set byte if carry (CF=1).
0F 94	SETE <i>r/m8</i>	M	Valid	Valid	Set byte if equal (ZF=1).
REX + 0F 94	SETE <i>r/m8</i> *	M	Valid	N.E.	Set byte if equal (ZF=1).
0F 9F	SETG <i>r/m8</i>	M	Valid	Valid	Set byte if greater (ZF=0 and SF=0F).
REX + 0F 9F	SETG <i>r/m8</i> *	M	Valid	N.E.	Set byte if greater (ZF=0 and SF=0F).
0F 9D	SETGE <i>r/m8</i>	M	Valid	Valid	Set byte if greater or equal (SF=0F).
REX + 0F 9D	SETGE <i>r/m8</i> *	M	Valid	N.E.	Set byte if greater or equal (SF=0F).
0F 9C	SETL <i>r/m8</i>	M	Valid	Valid	Set byte if less (SF≠ 0F).
REX + 0F 9C	SETL <i>r/m8</i> *	M	Valid	N.E.	Set byte if less (SF≠ 0F).
0F 9E	SETLE <i>r/m8</i>	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ 0F).
REX + 0F 9E	SETLE <i>r/m8</i> *	M	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ 0F).
0F 96	SETNA <i>r/m8</i>	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + 0F 96	SETNA <i>r/m8</i> *	M	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE <i>r/m8</i>	M	Valid	Valid	Set byte if not above or equal (CF=1).
REX + 0F 92	SETNAE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not above or equal (CF=1).
0F 93	SETNB <i>r/m8</i>	M	Valid	Valid	Set byte if not below (CF=0).
REX + 0F 93	SETNB <i>r/m8</i> *	M	Valid	N.E.	Set byte if not below (CF=0).
0F 97	SETNBE <i>r/m8</i>	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
REX + 0F 97	SETNBE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC <i>r/m8</i>	M	Valid	Valid	Set byte if not carry (CF=0).
REX + 0F 93	SETNC <i>r/m8</i> *	M	Valid	N.E.	Set byte if not carry (CF=0).
0F 95	SETNE <i>r/m8</i>	M	Valid	Valid	Set byte if not equal (ZF=0).
REX + 0F 95	SETNE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not equal (ZF=0).
0F 9E	SETNG <i>r/m8</i>	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ 0F)
REX + 0F 9E	SETNG <i>r/m8</i> *	M	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ 0F).
0F 9C	SETNGE <i>r/m8</i>	M	Valid	Valid	Set byte if not greater or equal (SF≠ 0F).
REX + 0F 9C	SETNGE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not greater or equal (SF≠ 0F).
0F 9D	SETNL <i>r/m8</i>	M	Valid	Valid	Set byte if not less (SF=0F).
REX + 0F 9D	SETNL <i>r/m8</i> *	M	Valid	N.E.	Set byte if not less (SF=0F).
0F 9F	SETNLE <i>r/m8</i>	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=0F).

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
REX + 0F 9F	SETNLE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=0F).
0F 91	SETNO <i>r/m8</i>	M	Valid	Valid	Set byte if not overflow (OF=0).
REX + 0F 91	SETNO <i>r/m8</i> *	M	Valid	N.E.	Set byte if not overflow (OF=0).
0F 9B	SETNP <i>r/m8</i>	M	Valid	Valid	Set byte if not parity (PF=0).
REX + 0F 9B	SETNP <i>r/m8</i> *	M	Valid	N.E.	Set byte if not parity (PF=0).
0F 99	SETNS <i>r/m8</i>	M	Valid	Valid	Set byte if not sign (SF=0).
REX + 0F 99	SETNS <i>r/m8</i> *	M	Valid	N.E.	Set byte if not sign (SF=0).
0F 95	SETNZ <i>r/m8</i>	M	Valid	Valid	Set byte if not zero (ZF=0).
REX + 0F 95	SETNZ <i>r/m8</i> *	M	Valid	N.E.	Set byte if not zero (ZF=0).
0F 90	SETO <i>r/m8</i>	M	Valid	Valid	Set byte if overflow (OF=1)
REX + 0F 90	SETO <i>r/m8</i> *	M	Valid	N.E.	Set byte if overflow (OF=1).
0F 9A	SETP <i>r/m8</i>	M	Valid	Valid	Set byte if parity (PF=1).
REX + 0F 9A	SETP <i>r/m8</i> *	M	Valid	N.E.	Set byte if parity (PF=1).
0F 9A	SETPE <i>r/m8</i>	M	Valid	Valid	Set byte if parity even (PF=1).
REX + 0F 9A	SETPE <i>r/m8</i> *	M	Valid	N.E.	Set byte if parity even (PF=1).
0F 9B	SETPO <i>r/m8</i>	M	Valid	Valid	Set byte if parity odd (PF=0).
REX + 0F 9B	SETPO <i>r/m8</i> *	M	Valid	N.E.	Set byte if parity odd (PF=0).
0F 98	SETS <i>r/m8</i>	M	Valid	Valid	Set byte if sign (SF=1).
REX + 0F 98	SETS <i>r/m8</i> *	M	Valid	N.E.	Set byte if sign (SF=1).
0F 94	SETZ <i>r/m8</i>	M	Valid	Valid	Set byte if zero (ZF=1).
REX + 0F 94	SETZ <i>r/m8</i> *	M	Valid	N.E.	Set byte if zero (ZF=1).

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

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

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

Many of the SET_{cc} instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals 0F. These alternate mnemonics are provided to make code more intelligible. Appendix B, “EFLAGS Condition Codes,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

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

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

Operation

IF condition
 THEN DEST \leftarrow 1;
 ELSE DEST \leftarrow 0;
 FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

SFENCE—Store Fence

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AE F8	SFENCE	NP	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes the SFENCE instruction in program order becomes globally visible before any store instruction that follows the SFENCE instruction. The SFENCE instruction is ordered with respect to store instructions, other SFENCE instructions, any LFENCE and MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form OF AE Fx, where x is in the range 8-F.

Operation

Wait_On_Following_Stores_Until(preceding_stores_globally_visible);

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_sfence(void)

Exceptions (All Operating Modes)

#UD If CPUID.01H:EDX.SSE[bit 25] = 0.
 If the LOCK prefix is used.

SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT <i>m</i>	M	Valid	Valid	Store GDTR to <i>m</i> .

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in bytes 3-5, and byte 6 is zero-filled. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In IA-32e mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386™ processors fill these bits with 0s.

Operation

IF instruction is SGDT

IF OperandSize = 16

THEN

DEST[0:15] ← GDTR(Limit);

DEST[16:39] ← GDTR(Base); (* 24 bits of base address stored *)

DEST[40:47] ← 0;

ELSE IF (32-bit Operand Size)

DEST[0:15] ← GDTR(Limit);

DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)

FI;

ELSE (* 64-bit Operand Size *)

DEST[0:15] ← GDTR(Limit);

DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)

FI;

FI;

Flags Affected

None.

Protected Mode Exceptions

#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If the destination is located in a non-writable 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.

Real-Address Mode Exceptions

#UD	If the destination operand is a register. If the LOCK prefix is used.
#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.

Virtual-8086 Mode Exceptions

#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
#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.

SHLD—Double Precision Shift Left

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF A4 /r ib	SHLD <i>r/m16, r16, imm8</i>	MRI	Valid	Valid	Shift <i>r/m16</i> to left <i>imm8</i> places while shifting bits from <i>r16</i> in from the right.
OF A5 /r	SHLD <i>r/m16, r16, CL</i>	MRC	Valid	Valid	Shift <i>r/m16</i> to left CL places while shifting bits from <i>r16</i> in from the right.
OF A4 /r ib	SHLD <i>r/m32, r32, imm8</i>	MRI	Valid	Valid	Shift <i>r/m32</i> to left <i>imm8</i> places while shifting bits from <i>r32</i> in from the right.
REX.W + OF A4 /r ib	SHLD <i>r/m64, r64, imm8</i>	MRI	Valid	N.E.	Shift <i>r/m64</i> to left <i>imm8</i> places while shifting bits from <i>r64</i> in from the right.
OF A5 /r	SHLD <i>r/m32, r32, CL</i>	MRC	Valid	Valid	Shift <i>r/m32</i> to left CL places while shifting bits from <i>r32</i> in from the right.
REX.W + OF A5 /r	SHLD <i>r/m64, r64, CL</i>	MRC	Valid	N.E.	Shift <i>r/m64</i> to left CL places while shifting bits from <i>r64</i> in from the right.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE

```

```

IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, SIZE - COUNT];
        (* Last bit shifted out on exit *)
        FOR i ← SIZE - 1 DOWN TO COUNT
            DO
                Bit(DEST, i) ← Bit(DEST, i - COUNT);
            OD;
        FOR i ← COUNT - 1 DOWN TO 0
            DO
                BIT[DEST, i] ← BIT[DEST, i - COUNT + SIZE];
            OD;
        FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

SHRD—Double Precision Shift Right

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AC /r ib	SHRD <i>r/m16, r16, imm8</i>	MRI	Valid	Valid	Shift <i>r/m16</i> to right <i>imm8</i> places while shifting bits from <i>r16</i> in from the left.
OF AD /r	SHRD <i>r/m16, r16, CL</i>	MRC	Valid	Valid	Shift <i>r/m16</i> to right CL places while shifting bits from <i>r16</i> in from the left.
OF AC /r ib	SHRD <i>r/m32, r32, imm8</i>	MRI	Valid	Valid	Shift <i>r/m32</i> to right <i>imm8</i> places while shifting bits from <i>r32</i> in from the left.
REX.W + OF AC /r ib	SHRD <i>r/m64, r64, imm8</i>	MRI	Valid	N.E.	Shift <i>r/m64</i> to right <i>imm8</i> places while shifting bits from <i>r64</i> in from the left.
OF AD /r	SHRD <i>r/m32, r32, CL</i>	MRC	Valid	Valid	Shift <i>r/m32</i> to right CL places while shifting bits from <i>r32</i> in from the left.
REX.W + OF AD /r	SHRD <i>r/m64, r64, CL</i>	MRC	Valid	N.E.	Shift <i>r/m64</i> to right CL places while shifting bits from <i>r64</i> in from the left.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE

```

```

IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, COUNT - 1]; (* Last bit shifted out on exit *)
        FOR i ← 0 TO SIZE - 1 - COUNT
            DO
                BIT[DEST, i] ← BIT[DEST, i + COUNT];
            OD;
        FOR i ← SIZE - COUNT TO SIZE - 1
            DO
                BIT[DEST, i] ← BIT[SRC, i + COUNT - SIZE];
            OD;
        FI;
    FI;
FI;

```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Moves either of the two packed double-precision floating-point values from destination operand (first operand) into the low quadword of the destination operand; moves either of the two packed double-precision floating-point values from the source operand into to the high quadword of the destination operand (see Figure 4-21). The select operand (third operand) determines which values are moved to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

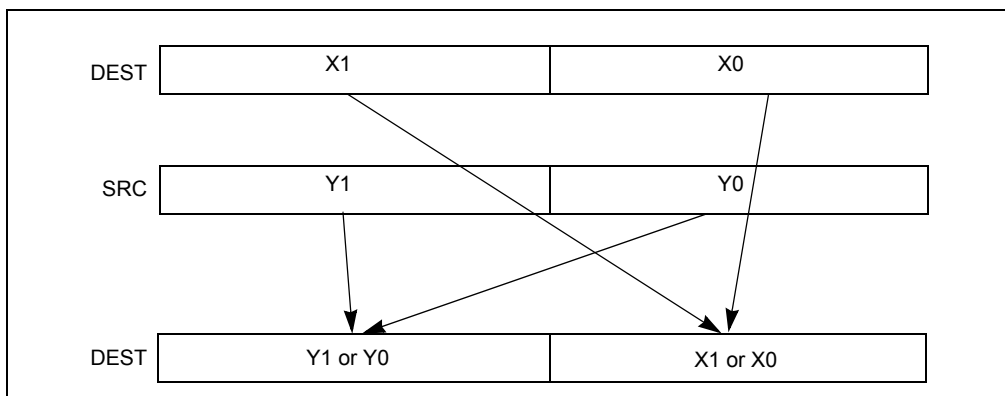


Figure 4-21. SHUFPD Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bit 0 selects which value is moved from the destination operand to the result (where 0 selects the low quadword and 1 selects the high quadword) and bit 1 selects which value is moved from the source operand to the result. Bits 2 through 7 of the select operand are reserved and must be set to 0.

Operation

```
IF SELECT[0] = 0
    THEN DEST[63:0] ← DEST[63:0];
    ELSE DEST[63:0] ← DEST[127:64]; FI;

IF SELECT[1] = 0
    THEN DEST[127:64] ← SRC[63:0];
    ELSE DEST[127:64] ← SRC[127:64]; FI;
```

SHUFPD (128-bit Legacy SSE version)

```
IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[VLMAX-1:128] (Unmodified)
```

VSHUFPD (VEX.128 encoded version)

```
IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[VLMAX-1:128] ← 0
```

VSHUFPD (VEX.256 encoded version)

```
IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMMO[2] = 0
    THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMMO[3] = 0
    THEN DEST[255:192] ← SRC2[191:128]
    ELSE DEST[255:192] ← SRC2[255:192] FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SHUFPD:    __m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)
VSHUFPD:    __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C6 /r ib SHUFPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Shuffle packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm1/m128</i> to <i>xmm1</i> .
VEX.NDS.128.OF.WIG C6 /r ib VSHUFPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG C6 /r ib VSHUFPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Moves two of the four packed single-precision floating-point values from the destination operand (first operand) into the low quadword of the destination operand; moves two of the four packed single-precision floating-point values from the source operand (second operand) into the high quadword of the destination operand (see Figure 4-22). The select operand (third operand) determines which values are moved to the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

determines which values are moved to the destination operand.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

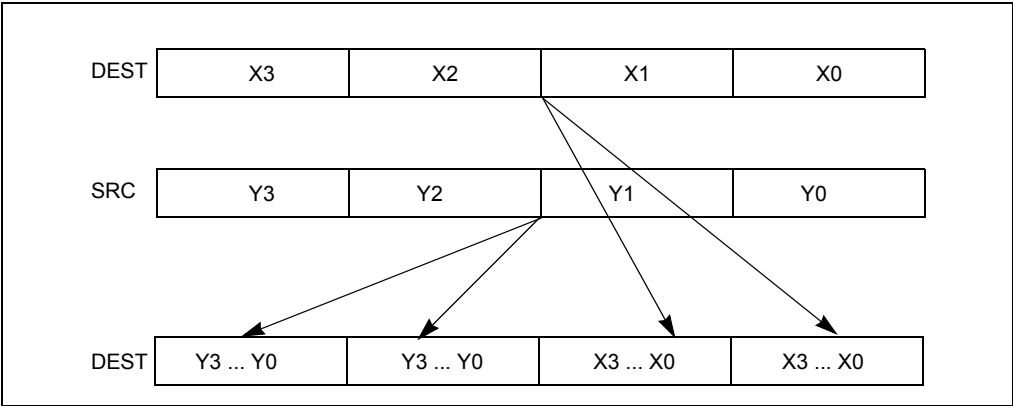


Figure 4-22. SHUFPS Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand to the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand to the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand to the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand to the high doubleword of the result.

Operation

CASE (SELECT[1:0]) OF

- 0: DEST[31:0] \leftarrow DEST[31:0];
- 1: DEST[31:0] \leftarrow DEST[63:32];
- 2: DEST[31:0] \leftarrow DEST[95:64];
- 3: DEST[31:0] \leftarrow DEST[127:96];

ESAC;

CASE (SELECT[3:2]) OF

- 0: DEST[63:32] \leftarrow DEST[31:0];
- 1: DEST[63:32] \leftarrow DEST[63:32];
- 2: DEST[63:32] \leftarrow DEST[95:64];
- 3: DEST[63:32] \leftarrow DEST[127:96];

ESAC;

CASE (SELECT[5:4]) OF

- 0: DEST[95:64] \leftarrow SRC[31:0];
- 1: DEST[95:64] \leftarrow SRC[63:32];
- 2: DEST[95:64] \leftarrow SRC[95:64];
- 3: DEST[95:64] \leftarrow SRC[127:96];

ESAC;

CASE (SELECT[7:6]) OF

- 0: DEST[127:96] \leftarrow SRC[31:0];
- 1: DEST[127:96] \leftarrow SRC[63:32];
- 2: DEST[127:96] \leftarrow SRC[95:64];
- 3: DEST[127:96] \leftarrow SRC[127:96];

ESAC;

SHUFPS (128-bit Legacy SSE version)

DEST[31:0] \leftarrow Select4(SRC1[127:0], imm8[1:0]);
 DEST[63:32] \leftarrow Select4(SRC1[127:0], imm8[3:2]);
 DEST[95:64] \leftarrow Select4(SRC2[127:0], imm8[5:4]);
 DEST[127:96] \leftarrow Select4(SRC2[127:0], imm8[7:6]);
 DEST[VLMAX-1:128] (Unmodified)

VSHUFPS (VEX.128 encoded version)

DEST[31:0] \leftarrow Select4(SRC1[127:0], imm8[1:0]);
 DEST[63:32] \leftarrow Select4(SRC1[127:0], imm8[3:2]);
 DEST[95:64] \leftarrow Select4(SRC2[127:0], imm8[5:4]);
 DEST[127:96] \leftarrow Select4(SRC2[127:0], imm8[7:6]);
 DEST[VLMAX-1:128] \leftarrow 0

VSHUFPS (VEX.256 encoded version)

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
 DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
 DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
 DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
 DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
 DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
 DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
 DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPS: __m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
 VSHUFPS: __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

SIDT—Store Interrupt Descriptor Table Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /1	SIDT <i>m</i>	M	Valid	Valid	Store IDTR to <i>m</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (<i>w</i>)	NA	NA	NA

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, if the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

Operation

IF instruction is SIDT

THEN

IF OperandSize = 16

THEN

DEST[0:15] ← IDTR(Limit);

DEST[16:39] ← IDTR(Base); (* 24 bits of base address stored; *)

DEST[40:47] ← 0;

ELSE IF (32-bit Operand Size)

DEST[0:15] ← IDTR(Limit);

DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)

ELSE (* 64-bit Operand Size *)

DEST[0:15] ← IDTR(Limit);

DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)

FI;

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the destination operand is a register. If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
#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.

SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /0	SLDT <i>r/m16</i>	M	Valid	Valid	Stores segment selector from LDTR in <i>r/m16</i> .
REX.W + OF 00 /0	SLDT <i>r64/m16</i>	M	Valid	Valid	Stores segment selector from LDTR in <i>r64/m16</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

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

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

Operation

DEST ← LDTR(SegmentSelector);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The SLDT instruction is not recognized in real-address mode.
-----	--

If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD The SLDT instruction is not recognized in virtual-8086 mode.
If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#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.
#UD If the LOCK prefix is used.

SMSW—Store Machine Status Word

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /4	SMSW <i>r/m16</i>	M	Valid	Valid	Store machine status word to <i>r/m16</i> .
OF 01 /4	SMSW <i>r32/m16</i>	M	Valid	Valid	Store machine status word in low-order 16 bits of <i>r32/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.
REX.W + OF 01 /4	SMSW <i>r64/m16</i>	M	Valid	Valid	Store machine status word in low-order 16 bits of <i>r64/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

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

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW *r16* operand size 16, store CR0[15:0] in *r16*
- SMSW *r32* operand size 32, zero-extend CR0[31:0], and store in *r32*
- SMSW *r64* operand size 64, zero-extend CR0[63:0], and store in *r64*
- SMSW *m16* operand size 16, store CR0[15:0] in *m16*
- SMSW *m16* operand size 32, store CR0[15:0] in *m16* (not *m32*)
- SMSW *m16* operands size 64, store CR0[15:0] in *m16* (not *m64*)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs. The is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

DEST ← CR0[15:0];
(* Machine status word *)

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Computes square roots of the packed double-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.66.0F.WIG 51 /r VSQRTPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in <i>xmm2/m128</i> and stores the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 51/r VSQRTPD <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in <i>ymm2/m256</i> and stores the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the square roots of the two packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

SQRTPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[VLMAX-1:128] (Unmodified)
```

VSQRTPD (VEX.128 encoded version)

```
DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[VLMAX-1:128] ← 0
```

VSQRTPD (VEX.256 encoded version) $\text{DEST}[63:0] \leftarrow \text{SQRT}(\text{SRC}[63:0])$ $\text{DEST}[127:64] \leftarrow \text{SQRT}(\text{SRC}[127:64])$ $\text{DEST}[191:128] \leftarrow \text{SQRT}(\text{SRC}[191:128])$ $\text{DEST}[255:192] \leftarrow \text{SQRT}(\text{SRC}[255:192])$ **Intel C/C++ Compiler Intrinsic Equivalent**SQRTPD: `__m128d _mm_sqrt_pd (m128d a)`SQRTPD: `__m256d _mm256_sqrt_pd (__m256d a);`**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv \neq 1111B.

SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 51 /r SQRTPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Computes square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.OF.WIG 51 /r VSQRTPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the result in <i>xmm1</i> .
VEX.256.OF.WIG 51/r VSQRTPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in <i>ymm2/m256</i> and stores the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the square roots of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

SQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[VLMAX-1:128] (Unmodified)
```

VSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

VSQRTPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SQRT}(\text{SRC}[31:0])$
 $\text{DEST}[63:32] \leftarrow \text{SQRT}(\text{SRC}[63:32])$
 $\text{DEST}[95:64] \leftarrow \text{SQRT}(\text{SRC}[95:64])$
 $\text{DEST}[127:96] \leftarrow \text{SQRT}(\text{SRC}[127:96])$
 $\text{DEST}[159:128] \leftarrow \text{SQRT}(\text{SRC}[159:128])$
 $\text{DEST}[191:160] \leftarrow \text{SQRT}(\text{SRC}[191:160])$
 $\text{DEST}[223:192] \leftarrow \text{SQRT}(\text{SRC}[223:192])$
 $\text{DEST}[255:224] \leftarrow \text{SQRT}(\text{SRC}[255:224])$

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPS: `__m128 _mm_sqrt_ps(__m128 a)`
 SQRTPS: `__m256 _mm256_sqrt_ps (__m256 a);`

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv \neq 1111B.

SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51 /r SQRTSD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE2	Computes square root of the low double-precision floating-point value in <i>xmm2/m64</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 51/r VSQRTSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m64</i>	RVM	V/V	AVX	Computes square root of the low double-precision floating point value in <i>xmm3/m64</i> and stores the results in <i>xmm2</i> . Also, upper double precision floating-point value (bits[127:64]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low double-precision floating-point value in the source operand (second operand) and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SQRTSD (128-bit Legacy SSE version)

DEST[63:0] ← SQRT(SRC[63:0])

DEST[VLMAX-1:64] (Unmodified)

VSQRTSD (VEX.128 encoded version)

DEST[63:0] ← SQRT(SRC2[63:0])

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD: `__m128d _mm_sqrt_sd (m128d a, m128d b)`

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Computes square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 51/r VSQRTSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Computes square root of the low single-precision floating-point value in <i>xmm3/m32</i> and stores the results in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order double-words of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SQRTSS (128-bit Legacy SSE version)

DEST[31:0] ← SQRT(SRC2[31:0])

DEST[VLMAX-1:32] (Unmodified)

VSQRTSS (VEX.128 encoded version)

DEST[31:0] ← SQRT(SRC2[31:0])

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSS: `__m128 _mm_sqrt_ss(__m128 a)`

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

STAC—Set AC Flag in EFLAGS Register

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF 01 CB	STAC	NP	Valid	Valid	Set the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register. This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

Operation

EFLAGS.AC ← 1;

Flags Affected

AC set. Other flags are unaffected.

Protected Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD
 If the LOCK prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

#UD The STAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

STC—Set Carry Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	NP	Valid	Valid	Set CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Sets the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

$CF \leftarrow 1$;

Flags Affected

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

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

STD—Set Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FD	STD	NP	Valid	Valid	Set DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

Operation

$DF \leftarrow 1$;

Flags Affected

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

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

STI—Set Interrupt Flag

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FB	STI	NP	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

If protected-mode virtual interrupts are not enabled, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized¹. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts (and SMIs) may be blocked for one macroinstruction following an STI.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-16 indicates the action of the STI instruction depending on the processor's mode of operation and the CPL/IOPL settings of the running program or procedure.

Operation is the same in all modes.

Table 4-16. Decision Table for STI Results

CR0.PE	EFLAGS.VM	EFLAGS.IOPL	CS.CPL	CR4.PVI	EFLAGS.VIP	CR4.VME	STI Result
0	X	X	X	X	X	X	IF = 1
1	0	≥ CPL	X	X	X	X	IF = 1
1	0	< CPL	3	1	X	X	VIF = 1
1	0	< CPL	< 3	X	X	X	GP Fault
1	0	< CPL	X	0	X	X	GP Fault
1	0	< CPL	X	X	1	X	GP Fault
1	1	3	X	X	X	X	IF = 1
1	1	< 3	X	X	0	1	VIF = 1
1	1	< 3	X	X	1	X	GP Fault
1	1	< 3	X	X	X	0	GP Fault

NOTES:

X = This setting has no impact.

- The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.

In the following instruction sequence, interrupts may be recognized before RET executes:

```
STI
STI
RET
```

Operation

```

IF PE = 0 (* Executing in real-address mode *)
  THEN
    IF ← 1; (* Set Interrupt Flag *)
  ELSE (* Executing in protected mode or virtual-8086 mode *)
    IF VM = 0 (* Executing in protected mode*)
      THEN
        IF IOPL ≥ CPL
          THEN
            IF ← 1; (* Set Interrupt Flag *)
          ELSE
            IF (IOPL < CPL) and (CPL = 3) and (PVI = 1)
              THEN
                VIF ← 1; (* Set Virtual Interrupt Flag *)
              ELSE
                #GP(0);
            FI;
          ELSE (* Executing in Virtual-8086 mode *)
            IF IOPL = 3
              THEN
                IF ← 1; (* Set Interrupt Flag *)
              ELSE
                IF ((IOPL < 3) and (VIP = 0) and (VME = 1))
                  THEN
                    VIF ← 1; (* Set Virtual Interrupt Flag *)
                  ELSE
                    #GP(0); (* Trap to virtual-8086 monitor *)
                FI;)
            FI;
          FI;
        FI;

```

Flags Affected

The IF flag is set to 1; or the VIF flag is set to 1. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF AE /3 STMXCSR <i>m32</i>	M	V/V	SSE	Store contents of MXCSR register to <i>m32</i> .
VEX.LZ.OF.WIG AE /3 VSTMXCSR <i>m32</i>	M	V/V	AVX	Store contents of MXCSR register to <i>m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

$m32 \leftarrow \text{MXCSR};$

Intel C/C++ Compiler Intrinsic Equivalent

`_mm_getcsr(void)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1,
 If VEX.vvvv ≠ 1111B.

STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AA	STOS <i>m8</i>	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS <i>m16</i>	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS <i>m32</i>	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS <i>m64</i>	NA	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	NA	Valid	N.E.	Store RAX at address RDI or EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NA	NA	NA	NA	NA

Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

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

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

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

NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

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

Operation

Non-64-bit Mode:

```

IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI - 1;
    FI;
  ELSE IF (Word store)
    THEN
      DEST ← AX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 2;
        ELSE (E)DI ← (E)DI - 2;
      FI;
    FI;
  ELSE IF (Doubleword store)
    THEN
      DEST ← EAX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI - 4;
      FI;
    FI;
  FI;

```

64-bit Mode:

```

IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (R|E)DI ← (R|E)DI + 1;
      ELSE (R|E)DI ← (R|E)DI - 1;
    FI;
  ELSE IF (Word store)

```

```

THEN
    DEST ← AX;
    THEN IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 2;
        ELSE (R|E)DI ← (R|E)DI - 2;
    FI;
FI;
ELSE IF (Doubleword store)
    THEN
        DEST ← EAX;
        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 4;
            ELSE (R|E)DI ← (R|E)DI - 4;
        FI;
    FI;
ELSE IF (Quadword store using REX.W )
    THEN
        DEST ← RAX;
        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 8;
            ELSE (R|E)DI ← (R|E)DI - 8;
        FI;
    FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a NULL segment selector.
#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the ES segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

STR—Store Task Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /1	STR <i>r/m16</i>	M	Valid	Valid	Stores segment selector from TR in <i>r/m16</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (w)	NA	NA	NA

Description

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

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

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

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

Operation

DEST ← TR(SegmentSelector);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the 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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The STR instruction is not recognized in real-address mode.
-----	---

Virtual-8086 Mode Exceptions

#UD	The STR instruction is not recognized in virtual-8086 mode.
-----	---

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

SUB—Subtract

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
2C <i>ib</i>	SUB AL, <i>imm8</i>	I	Valid	Valid	Subtract <i>imm8</i> from AL.
2D <i>iw</i>	SUB AX, <i>imm16</i>	I	Valid	Valid	Subtract <i>imm16</i> from AX.
2D <i>id</i>	SUB EAX, <i>imm32</i>	I	Valid	Valid	Subtract <i>imm32</i> from EAX.
REX.W + 2D <i>id</i>	SUB RAX, <i>imm32</i>	I	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from RAX.
80 /5 <i>ib</i>	SUB <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Subtract <i>imm8</i> from <i>r/m8</i> .
REX + 80 /5 <i>ib</i>	SUB <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract <i>imm8</i> from <i>r/m8</i> .
81 /5 <i>iw</i>	SUB <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Subtract <i>imm16</i> from <i>r/m16</i> .
81 /5 <i>id</i>	SUB <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Subtract <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /5 <i>id</i>	SUB <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from <i>r/m64</i> .
83 /5 <i>ib</i>	SUB <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /5 <i>ib</i>	SUB <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /5 <i>ib</i>	SUB <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract sign-extended <i>imm8</i> from <i>r/m64</i> .
28 /r	SUB <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Subtract <i>r8</i> from <i>r/m8</i> .
REX + 28 /r	SUB <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Subtract <i>r8</i> from <i>r/m8</i> .
29 /r	SUB <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Subtract <i>r16</i> from <i>r/m16</i> .
29 /r	SUB <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Subtract <i>r32</i> from <i>r/m32</i> .
REX.W + 29 /r	SUB <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Subtract <i>r64</i> from <i>r/m64</i> .
2A /r	SUB <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Subtract <i>r/m8</i> from <i>r8</i> .
REX + 2A /r	SUB <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Subtract <i>r/m8</i> from <i>r8</i> .
2B /r	SUB <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Subtract <i>r/m16</i> from <i>r16</i> .
2B /r	SUB <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Subtract <i>r/m32</i> from <i>r32</i> .
REX.W + 2B /r	SUB <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Subtract <i>r/m64</i> from <i>r64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/26/32</i>	NA	NA
MI	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	<i>imm8/26/32</i>	NA	NA
MR	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	ModRM: <i>reg</i> (<i>r</i>)	NA	NA
RM	ModRM: <i>reg</i> (<i>r</i> , <i>w</i>)	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA

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 an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

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

Operation

$DEST \leftarrow (DEST - SRC);$

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed double-precision floating-point values in <i>xmm2/m128</i> from <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed double-precision floating-point values in <i>xmm3/mem</i> from <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Subtract packed double-precision floating-point values in <i>ymm3/mem</i> from <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the two packed double-precision floating-point values in the source operand (second operand) from the two packed double-precision floating-point values in the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

SUBPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] - SRC[63:0]
 DEST[127:64] ← DEST[127:64] - SRC[127:64]
 DEST[VLMAX-1:128] (Unmodified)

VSUBPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
 DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
 DEST[VLMAX-1:128] ← 0

VSUBPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] - \text{SRC2}[127:64]$

$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] - \text{SRC2}[191:128]$

$\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] - \text{SRC2}[255:192]$

Intel C/C++ Compiler Intrinsic Equivalent

SUBPD: `__m128d _mm_sub_pd (m128d a, m128d b)`

VSUBPD: `__m256d _mm256_sub_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5C /r SUBPS <i>xmm1</i> <i>xmm2/m128</i>	RM	V/V	SSE	Subtract packed single-precision floating-point values in <i>xmm2/mem</i> from <i>xmm1</i> .
VEX.NDS.128.OF.WIG 5C /r VSUBPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed single-precision floating-point values in <i>xmm3/mem</i> from <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.OF.WIG 5C /r VSUBPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Subtract packed single-precision floating-point values in <i>ymm3/mem</i> from <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the four packed single-precision floating-point values in the source operand (second operand) from the four packed single-precision floating-point values in the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

SUBPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VSUBPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VSUBPS (VEX.256 encoded version)

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$$

$$\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$$

$$\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$$

$$\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$$

$$\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC2}[159:128]$$

$$\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] - \text{SRC2}[191:160]$$

$$\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] - \text{SRC2}[223:192]$$

$$\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] - \text{SRC2}[255:224]$$
Intel C/C++ Compiler Intrinsic Equivalent

SUBPS: `__m128 _mm_sub_ps(__m128 a, __m128 b)`

VSUBPS: `__m256 _mm256_sub_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

SUBSD—Subtract Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD <i>xmm1</i> , <i>xmm2</i> / <i>m64</i>	RM	V/V	SSE2	Subtracts the low double-precision floating-point values in <i>xmm2/mem64</i> from <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5C /r VSUBSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i> / <i>m64</i>	RVM	V/V	AVX	Subtract the low double-precision floating-point value in <i>xmm3/mem</i> from <i>xmm2</i> and store the result in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the low double-precision floating-point value in the source operand (second operand) from the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SUBSD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)

VSUBSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SUBSD: `__m128d _mm_sub_sd (m128d a, m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Subtract the lower single-precision floating-point values in <i>xmm2/m32</i> from <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5C /r VSUBSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Subtract the low single-precision floating-point value in <i>xmm3/mem</i> from <i>xmm2</i> and store the result in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the low single-precision floating-point value in the source operand (second operand) from the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SUBSS (128-bit Legacy SSE version)

DEST[31:0] ← DEST[31:0] - SRC[31:0]

DEST[VLMAX-1:32] (Unmodified)

VSUBSS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SUBSS: `__m128 _mm_sub_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 F8	SWAPGS	NP	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

Operation

IF CS.L \neq 1 (* Not in 64-Bit Mode *)

THEN

#UD; FI;

IF CPL \neq 0

THEN #GP(0); FI;

tmp \leftarrow GS.base;

GS.base \leftarrow IA32_KERNEL_GS_BASE;

IA32_KERNEL_GS_BASE \leftarrow tmp;

Flags Affected

None

Protected Mode Exceptions

#UD If Mode \neq 64-Bit.

Real-Address Mode Exceptions

#UD If Mode \neq 64-Bit.

Virtual-8086 Mode Exceptions

#UD If Mode \neq 64-Bit.

Compatibility Mode Exceptions

#UD If Mode ≠ 64-Bit.

64-Bit Mode Exceptions

#GP(0) If CPL ≠ 0.
 If the LOCK prefix is used.

SYSCALL—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 05	SYSCALL	NP	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

Operation

IF (CS.L \neq 1) or (IA32_EFER.LMA \neq 1) or (IA32_EFER.SCE \neq 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)

THEN #UD;

FI;

RCX \leftarrow RIP; (* Will contain address of next instruction *)

RIP \leftarrow IA32_LSTAR;

R11 \leftarrow RFLAGS;

RFLAGS \leftarrow RFLAGS AND NOT(IA32_FMASK);

CS.Selector \leftarrow IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)

(* Set rest of CS to a fixed value *)

CS.Base \leftarrow 0; (* Flat segment *)

CS.Limit \leftarrow FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)

CS.Type \leftarrow 11; (* Execute/read code, accessed *)

CS.S \leftarrow 1;

CS.DPL \leftarrow 0;

CS.P \leftarrow 1;

CS.L \leftarrow 1; (* Entry is to 64-bit mode *)

CS.D \leftarrow 0; (* Required if CS.L = 1 *)

CS.G \leftarrow 1; (* 4-KByte granularity *)

CPL \leftarrow 0;

SS.Selector \leftarrow IA32_STAR[47:32] + 8;	(* SS just above CS *)
(* Set rest of SS to a fixed value *)	
SS.Base \leftarrow 0;	(* Flat segment *)
SS.Limit \leftarrow FFFFFFFH;	(* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type \leftarrow 3;	(* Read/write data, accessed *)
SS.S \leftarrow 1;	
SS.DPL \leftarrow 0;	
SS.P \leftarrow 1;	
SS.B \leftarrow 1;	(* 32-bit stack segment *)
SS.G \leftarrow 1;	(* 4-KByte granularity *)

Flags Affected

All.

Protected Mode Exceptions

#UD The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The SYSCALL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The SYSCALL instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The SYSCALL instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If IA32_EFER.SCE = 0.
 If the LOCK prefix is used.

SYSENTER—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 34	SYSENTER	NP	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
    FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM ← 0;                (* Ensures protected mode execution *)
RFLAGS.IF ← 0;                (* Mask interrupts *)
IF in IA-32e mode
    THEN
        RSP ← IA32_SYSENTER_ESP;
        RIP ← IA32_SYSENTER_EIP;
    ELSE
        ESP ← IA32_SYSENTER_ESP[31:0];
        EIP ← IA32_SYSENTER_EIP[31:0];
    FI;

CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
                                (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                   (* Flat segment *)
CS.Limit ← FFFFFFFH;           (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                  (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
IF in IA-32e mode
    THEN
        CS.L ← 1;              (* Entry is to 64-bit mode *)
        CS.D ← 0;              (* Required if CS.L = 1 *)
    ELSE
        CS.L ← 0;
        CS.D ← 1;              (* 32-bit code segment*)
    FI;
CS.G ← 1;                      (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8;  (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                   (* Flat segment *)
SS.Limit ← FFFFFFFH;           (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                   (* Read/write data, accessed *)
```

$SS.S \leftarrow 1;$
 $SS.DPL \leftarrow 0;$
 $SS.P \leftarrow 1;$
 $SS.B \leftarrow 1;$ (* 32-bit stack segment*)
 $SS.G \leftarrow 1;$ (* 4-KByte granularity *)

Flags Affected

VM, IF (see Operation above)

Protected Mode Exceptions

#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP The SYSENTER instruction is not recognized in real-address mode.
 #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 35	SYSEXIT	NP	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + OF 35	SYSEXIT	NP	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protection levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
    FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;

IF operand size is 64-bit

THEN (* Return to 64-bit mode *)

RSP ← RCX;

RIP ← RDX;

ELSE (* Return to protected mode or compatibility mode *)

RSP ← ECX;

RIP ← EDX;

FI;

IF operand size is 64-bit (* Operating system provides CS; RPL forced to 3 *)

THEN CS.Selector ← IA32_SYSENTER_CS[15:0] + 32;

ELSE CS.Selector ← IA32_SYSENTER_CS[15:0] + 16;

FI;

CS.Selector ← CS.Selector OR 3; (* RPL forced to 3 *)

(* Set rest of CS to a fixed value *)

CS.Base ← 0; (* Flat segment *)

CS.Limit ← FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)

CS.Type ← 11; (* Execute/read code, accessed *)

CS.S ← 1;

CS.DPL ← 3;

CS.P ← 1;

IF operand size is 64-bit

THEN (* return to 64-bit mode *)

CS.L ← 1; (* 64-bit code segment *)

CS.D ← 0; (* Required if CS.L = 1 *)

ELSE (* return to protected mode or compatibility mode *)

CS.L ← 0;

CS.D ← 1; (* 32-bit code segment*)

FI;

CS.G ← 1; (* 4-KByte granularity *)

CPL ← 3;

SS.Selector ← CS.Selector + 8; (* SS just above CS *)

(* Set rest of SS to a fixed value *)

SS.Base ← 0; (* Flat segment *)

SS.Limit ← FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)

SS.Type ← 3; (* Read/write data, accessed *)

SS.S ← 1;

SS.DPL ← 3;

SS.P ← 1;

SS.B ← 1; (* 32-bit stack segment*)

SS.G ← 1; (* 4-KByte granularity *)

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If IA32_SYSENTER_CS[15:2] = 0.

If CPL ≠ 0.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP The SYSEXIT instruction is not recognized in real-address mode.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) The SYSEXIT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If IA32_SYSENTER_CS = 0.
 If CPL ≠ 0.
 If RCX or RDX contains a non-canonical address.
- #UD If the LOCK prefix is used.

SYSRET—Return From Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 07	SYSRET	NP	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + OF 07	SYSRET	NP	Valid	Invalid	Return to 64-bit mode from fast system call

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.¹ With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, “Interrupt Stack Table,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
 - Confirming that the value of RCX is canonical before executing SYSRET.
 - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
 - Using the IST mechanism for gate 13 (#GP) in the IDT.

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
  THEN #UD; FI;
IF (CPL ≠ 0) OR (RCX is not canonical) THEN #GP(0); FI;
```

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.


```

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;          (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
FI;
CS.Selector ← CS.Selector OR 3;          (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                          (* Flat segment *)
CS.Limit ← FFFFFFFH;                  (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                        (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L ← 1;                    (* 64-bit code segment *)
        CS.D ← 0;                    (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L ← 0;                    (* Compatibility mode *)
        CS.D ← 1;                    (* 32-bit code segment *)
FI;
CS.G ← 1;                             (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3; (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                          (* Flat segment *)
SS.Limit ← FFFFFFFH;                  (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                          (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                             (* 32-bit stack segment *)
SS.G ← 1;                             (* 4-KByte granularity *)

```

Flags Affected

All.

Protected Mode Exceptions

#UD The SYSRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The SYSRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The SYSRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The SYSRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If IA32_EFER.SCE = 0.
 If the LOCK prefix is used.

#GP(0) If CPL ≠ 0.
 If RCX contains a non-canonical address.

TEST—Logical Compare

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	I	Valid	Valid	AND <i>imm8</i> with AL; set SF, ZF, PF according to result.
A9 <i>iw</i>	TEST AX, <i>imm16</i>	I	Valid	Valid	AND <i>imm16</i> with AX; set SF, ZF, PF according to result.
A9 <i>id</i>	TEST EAX, <i>imm32</i>	I	Valid	Valid	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result.
REX.W + A9 <i>id</i>	TEST RAX, <i>imm32</i>	I	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 <i>ib</i>	TEST <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + F6 /0 <i>ib</i>	TEST <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
F7 /0 <i>iw</i>	TEST <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	AND <i>imm16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
F7 /0 <i>id</i>	TEST <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	AND <i>imm32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + F7 /0 <i>id</i>	TEST <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with <i>r/m64</i> ; set SF, ZF, PF according to result.
84 / <i>r</i>	TEST <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + 84 / <i>r</i>	TEST <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	AND <i>r16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	AND <i>r32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + 85 / <i>r</i>	TEST <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	AND <i>r64</i> with <i>r/m64</i> ; set SF, ZF, PF according to result.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/16/32</i>	NA	NA
MI	ModRM: <i>r/m</i> (<i>r</i>)	<i>imm8/16/32</i>	NA	NA
MR	ModRM: <i>r/m</i> (<i>r</i>)	ModRM: <i>reg</i> (<i>r</i>)	NA	NA

Description

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

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

TEMP \leftarrow SRC1 AND SRC2;
SF \leftarrow MSB(TEMP);

IF TEMP = 0
 THEN ZF \leftarrow 1;
 ELSE ZF \leftarrow 0;

FI;

PF \leftarrow BitwiseXNOR(TEMP[0:7]);
CF \leftarrow 0;
OF \leftarrow 0;
(* AF is undefined *)

Flags Affected

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

Protected Mode Exceptions

#GP(0)	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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

TZCNT — Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 OF BC /r TZCNT r16, r/m16	RM	V/V	BMI1	Count the number of trailing zero bits in <i>r/m16</i> , return result in <i>r16</i> .
F3 OF BC /r TZCNT r32, r/m32	RM	V/V	BMI1	Count the number of trailing zero bits in <i>r/m32</i> , return result in <i>r32</i> .
REX.W + F3 OF BC /r TZCNT r64, r/m64	RM	V/N.E.	BMI1	Count the number of trailing zero bits in <i>r/m64</i> , return result in <i>r64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

```
temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )
```

```
    temp ← temp + 1
    DEST ← DEST + 1
OD
```

```
IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
```

```
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

TZCNT: unsigned __int32 _tzcnt_u32(unsigned __int32 src);

TZCNT: unsigned __int64 _tzcnt_u64(unsigned __int64 src);

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#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(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 2E /r UCOMISD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE2	Compares (unordered) the low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m64</i> and set the EFLAGS accordingly.
VEX.LIG.66.OF.WIG 2E /r VUCOMISD <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Compare low double precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Performs an unordered compare of the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RESULT \leftarrow UnorderedCompare(SRC1[63:0] < > SRC2[63:0]) {

(* Set EFLAGS *)

CASE (RESULT) OF

UNORDERED: ZF, PF, CF \leftarrow 111;

GREATER_THAN: ZF, PF, CF \leftarrow 000;

LESS_THAN: ZF, PF, CF \leftarrow 001;

EQUAL: ZF, PF, CF \leftarrow 100;

ESAC;

OF, AF, SF \leftarrow 0;

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_ucomieq_sd(__m128d a, __m128d b)

int _mm_ucomilt_sd(__m128d a, __m128d b)

int _mm_ucomile_sd(__m128d a, __m128d b)

int _mm_ucomigt_sd(__m128d a, __m128d b)

int _mm_ucomige_sd(__m128d a, __m128d b)

int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv \neq 1111B.

UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 2E /r UCOMISS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Compare lower single-precision floating-point value in <i>xmm1</i> register with lower single-precision floating-point value in <i>xmm2/mem</i> and set the status flags accordingly.
VEX.LIG.OF.WIG 2E /r VUCOMISS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	AVX	Compare low single precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Performs an unordered compare of the single-precision floating-point values in the low doublewords of the source operand 1 (first operand) and the source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that -0.0 is equal to $+0.0$.

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISS instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RESULT \leftarrow UnorderedCompare(SRC1[31:0] <> SRC2[31:0]) {

(* Set EFLAGS *)

CASE (RESULT) OF

UNORDERED: ZF,PF,CF \leftarrow 111;

GREATER_THAN: ZF,PF,CF \leftarrow 000;

LESS_THAN: ZF,PF,CF \leftarrow 001;

EQUAL: ZF,PF,CF \leftarrow 100;

ESAC;

OF,AF,SF \leftarrow 0;

Intel C/C++ Compiler Intrinsic Equivalent

int __mm_ucomieq_ss(__m128 a, __m128 b)

int __mm_ucomilt_ss(__m128 a, __m128 b)

int __mm_ucomile_ss(__m128 a, __m128 b)

int __mm_ucomigt_ss(__m128 a, __m128 b)

int __mm_ucomige_ss(__m128 a, __m128 b)

int __mm_ucomineq_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv ≠ 1111B.

UD2—Undefined Instruction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 0B	UD2	NP	Valid	Valid	Raise invalid opcode exception.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD2 instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD2 instruction (and not the following instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

#UD (* Generates invalid opcode exception *);

Flags Affected

None.

Exceptions (All Operating Modes)

#UD Raises an invalid opcode exception in all operating modes.

UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 15 /r UNPCKHPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.OF.WIG 15 /r VUNPCKHPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.66.OF.WIG 15 /r VUNPCKHPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-23.

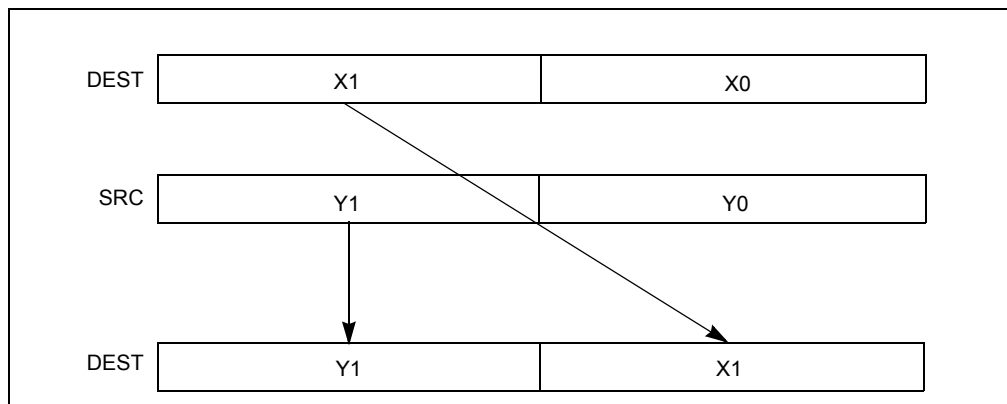


Figure 4-23. UNPCKHPD Instruction High Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

UNPCKHPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$

$\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$

$\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VUNPCKHPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$

$\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VUNPCKHPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[127:64]$

$\text{DEST}[127:64] \leftarrow \text{SRC2}[127:64]$

$\text{DEST}[191:128] \leftarrow \text{SRC1}[255:192]$

$\text{DEST}[255:192] \leftarrow \text{SRC2}[255:192]$

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD: `__m128d _mm_unpackhi_pd(__m128d a, __m128d b)`

UNPCKHPD: `__m256d _mm256_unpackhi_pd(__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 15 /r UNPCKHPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .
VEX.NDS.128.OF.WIG 15 /r VUNPCKHPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.OF.WIG 15 /r VUNPCKHPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-24. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

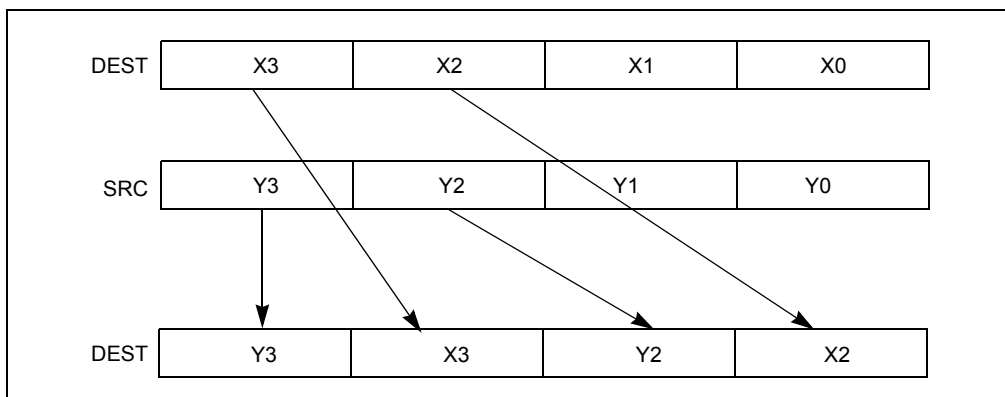


Figure 4-24. UNPCKHPS Instruction High Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8–XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

UNPCKHPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[95:64]$
 $\text{DEST}[63:32] \leftarrow \text{SRC2}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[127:96]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VUNPCKHPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[95:64]$
 $\text{DEST}[63:32] \leftarrow \text{SRC2}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[127:96]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VUNPCKHPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[95:64]$
 $\text{DEST}[63:32] \leftarrow \text{SRC2}[95:64]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[127:96]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[223:192]$
 $\text{DEST}[191:160] \leftarrow \text{SRC2}[223:192]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[255:224]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS: `__m128 _mm_unpackhi_ps(__m128 a, __m128 b)`
 UNPCKHPS: `__m256 _mm256_unpackhi_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 14 /r UNPCKLPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.OF.WIG 14 /r VUNPCKLPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.66.OF.WIG 14 /r VUNPCKLPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-25. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

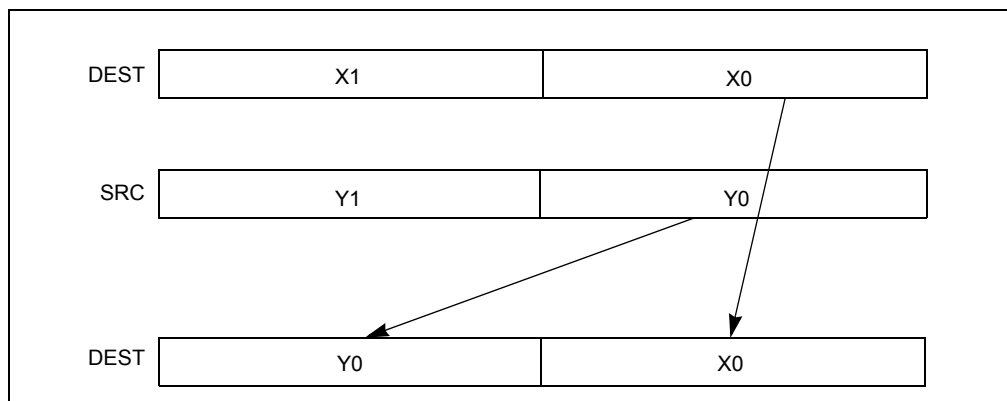


Figure 4-25. UNPCKLPD Instruction Low Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

UNPCKLPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]$

$\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VUNPCKLPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]$

$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VUNPCKLPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]$

$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128]$

$\text{DEST}[255:192] \leftarrow \text{SRC2}[191:128]$

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD: `__m128d _mm_unpacklo_pd(__m128d a, __m128d b)`

UNPCKLPD: `__m256d _mm256_unpacklo_pd(__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 14 /r UNPCKLPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .
VEX.NDS.128.OF.WIG 14 /r VUNPCKLPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>xmm2</i> and <i>xmm3/m128</i> .
VEX.NDS.256.OF.WIG 14 /r VUNPCKLPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of <i>ymm2</i> and <i>ymm3/m256</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-26. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

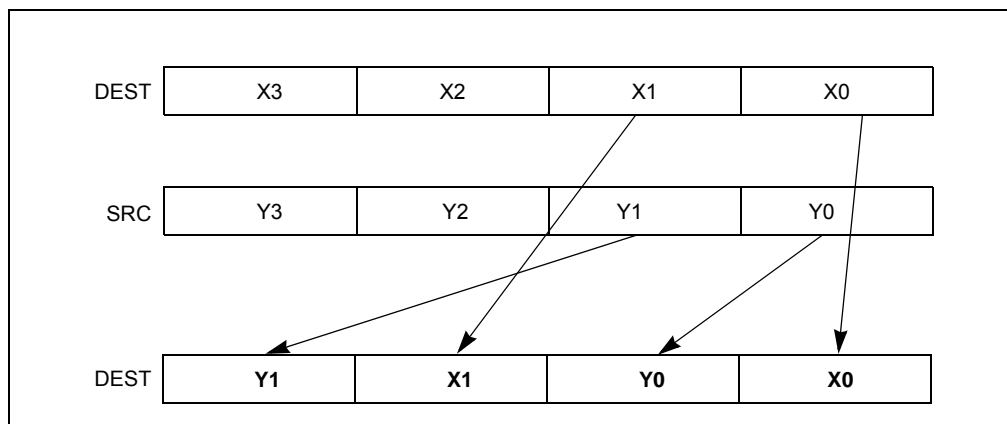


Figure 4-26. UNPCKLPS Instruction Low Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

UNPCKLPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC2}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[63:32]$
 $\text{DEST}[\text{VLMAX}-1:128]$ (Unmodified)

VUNPCKLPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC2}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[63:32]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VUNPCKLPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC2}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[63:32]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC2}[159:128]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[191:160]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[191:160]$

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS: `__m128 _mm_unpacklo_ps(__m128 a, __m128 b)`

UNPCKLPS: `__m256 _mm256_unpacklo_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VBROADCAST—Broadcast Floating-Point Data

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS <i>xmm1</i> , <i>m32</i>	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 18 /r VBROADCASTSS <i>ymm1</i> , <i>m32</i>	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 19 /r VBROADCASTSD <i>ymm1</i> , <i>m64</i>	RM	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 <i>ymm1</i> , <i>m128</i>	RM	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in <i>ymm1</i> .
VEX.128.66.0F38.W0 18/r VBROADCASTSS <i>xmm1</i> , <i>xmm2</i>	RM	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 18 /r VBROADCASTSS <i>ymm1</i> , <i>xmm2</i>	RM	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 19 /r VBROADCASTSD <i>ymm1</i> , <i>xmm2</i>	RM	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Load floating point values from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

VBROADCASTSD and VBROADCASTF128 are only supported as 256-bit wide versions. VBROADCASTSS is supported in both 128-bit and 256-bit wide versions.

Memory and register source operand syntax support of 256-bit instructions depend on the processor's enumeration of the following conditions with respect to CPUID.1:ECX.AVX[bit 28] and CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]:

- If CPUID.1:ECX.AVX = 1 and CPUID.(EAX=07H, ECX=0H):EBX.AVX2 = 0: the destination operand is a YMM register. The source operand support can be either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD.
- If CPUID.1:ECX.AVX = 1 and CPUID.(EAX=07H, ECX=0H):EBX.AVX2 = 1: the destination operand is a YMM register. The source operand support can be a register or memory location.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. An attempt to execute VBROADCASTSD or VBROADCASTF128 encoded with VEX.L = 0 will cause an #UD exception. Attempts to execute any VBROADCAST* instruction with VEX.W = 1 will cause #UD.

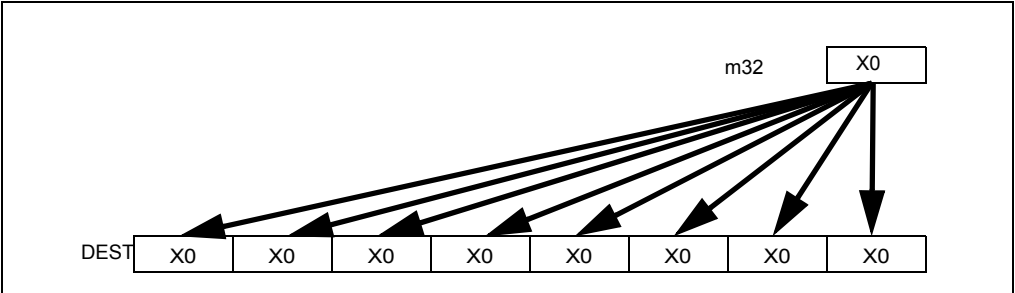


Figure 4-27. VBROADCASTSS Operation (VEX.256 encoded version)

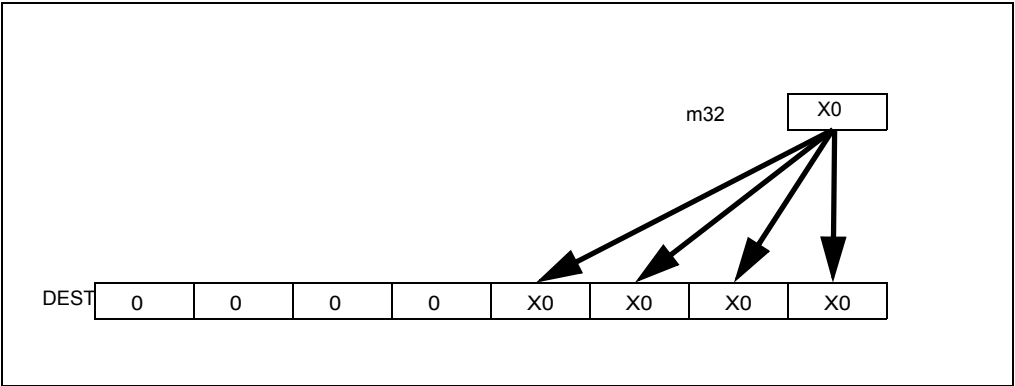


Figure 4-28. VBROADCASTSS Operation (128-bit version)

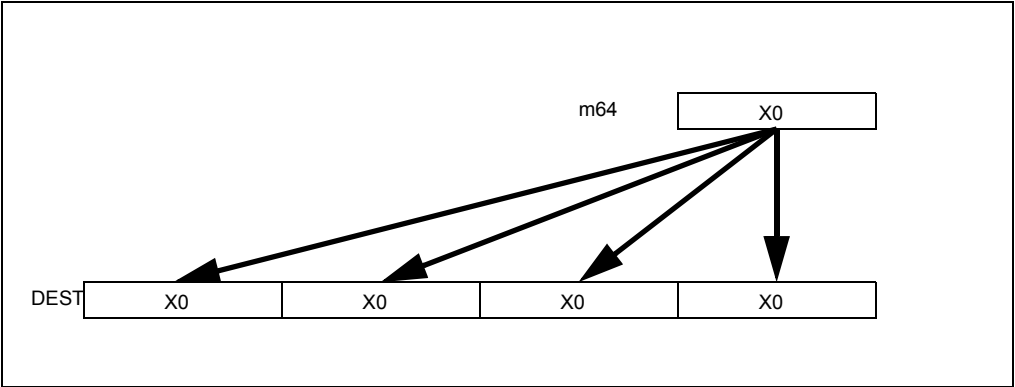


Figure 4-29. VBROADCASTSD Operation

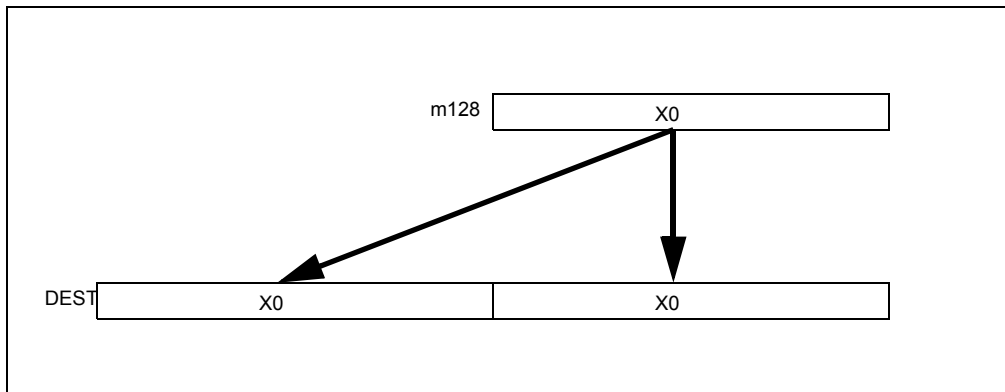


Figure 4-30. VBROADCASTF128 Operation

Operation

VBROADCASTSS (128 bit version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[VLMAX-1:128] ← 0
```

VBROADCASTSS (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
```

VBROADCASTSD (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
```

VBROADCASTF128

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[VLMAX-1:128] ← temp
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VBROADCASTSS:    __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS:    __m256 _mm256_broadcast_ss(float *a);
VBROADCASTSD:    __m256d _mm256_broadcast_sd(double *a);
```

VBROADCASTF128: `__m256 _mm256_broadcast_ps(__m128 * a);`

VBROADCASTF128: `__m256d _mm256_broadcast_pd(__m128d * a);`

Flags Affected

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0 for VBROADCASTSD,
	If VEX.L = 0 for VBROADCASTF128,
	If VEX.W = 1.

VCVTPH2PS—Convert 16-bit FP Values to Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 13 /r VCVTPH2PS <i>ymm1, xmm2/m128</i>	RM	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in <i>xmm2/m128</i> to packed single-precision floating-point value in <i>ymm1</i> .
VEX.128.66.0F38.W0 13 /r VCVTPH2PS <i>xmm1, xmm2/m64</i>	RM	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in <i>xmm2/m64</i> to packed single-precision floating-point value in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four/eight packed half precision (16-bits) floating-point values in the low-order 64/128 bits of an XMM/YMM register or 64/128-bit memory location to four/eight packed single-precision floating-point values and writes the converted values into the destination XMM/YMM register.

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

128-bit version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (VLMAX-1:128) of the corresponding destination YMM register are zeroed.

256-bit version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv is reserved (must be 1111b).

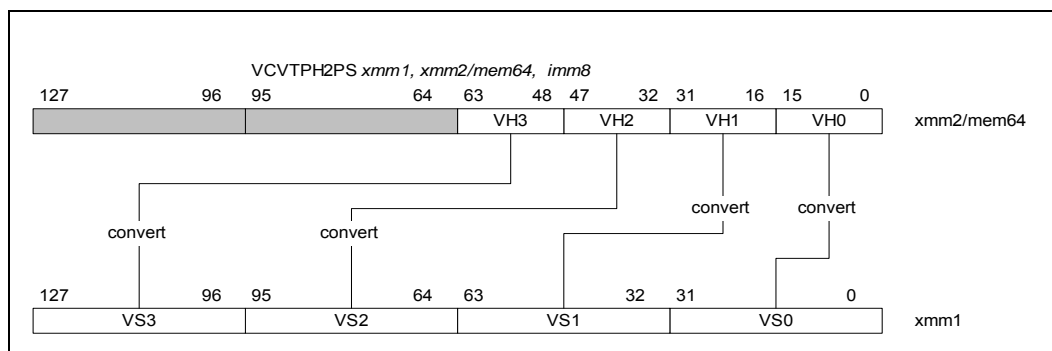


Figure 4-31. VCVTPH2PS (128-bit Version)

Operation

```

vCvt_h2s(SRC1[15:0])
{
    RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}

```


VCVTPH2PS (VEX.256 encoded version)

$DEST[31:0] \leftarrow vCvt_h2s(SRC1[15:0]);$
 $DEST[63:32] \leftarrow vCvt_h2s(SRC1[31:16]);$
 $DEST[95:64] \leftarrow vCvt_h2s(SRC1[47:32]);$
 $DEST[127:96] \leftarrow vCvt_h2s(SRC1[63:48]);$
 $DEST[159:128] \leftarrow vCvt_h2s(SRC1[79:64]);$
 $DEST[191:160] \leftarrow vCvt_h2s(SRC1[95:80]);$
 $DEST[223:192] \leftarrow vCvt_h2s(SRC1[111:96]);$
 $DEST[255:224] \leftarrow vCvt_h2s(SRC1[127:112]);$

VCVTPH2PS (VEX.128 encoded version)

$DEST[31:0] \leftarrow vCvt_h2s(SRC1[15:0]);$
 $DEST[63:32] \leftarrow vCvt_h2s(SRC1[31:16]);$
 $DEST[95:64] \leftarrow vCvt_h2s(SRC1[47:32]);$
 $DEST[127:96] \leftarrow vCvt_h2s(SRC1[63:48]);$
 $DEST[VLMAX-1:128] \leftarrow 0$

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

`__m128 _mm_cvtph_ps (__m128i m1);`
`__m256 _mm256_cvtph_ps (__m128i m1)`

SIMD Floating-Point Exceptions

Invalid

Other Exceptions

Exceptions Type 11 (do not report #AC); additionally
 #UD If VEX.W=1.

VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH <i>xmm1/m128, ymm2, imm8</i>	MR	V/V	F16C	Convert eight packed single-precision floating-point value in <i>ymm2</i> to packed half-precision (16-bit) floating-point value in <i>xmm1/mem</i> . <i>Imm8</i> provides rounding controls.
VEX.128.66.0F3A.W0.1D /r ib VCVTPS2PH <i>xmm1/m64, xmm2, imm8</i>	MR	V/V	F16C	Convert four packed single-precision floating-point value in <i>xmm2</i> to packed half-precision (16-bit) floating-point value in <i>xmm1/mem</i> . <i>Imm8</i> provides rounding controls.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Convert four or eight packed single-precision floating values in first source operand to four or eight packed half-precision (16-bit) floating-point values. The rounding mode is specified using the immediate field (*imm8*).

Non-zero tiny results are converted to zero, denormals, or the smallest normalized half-precision floating-point value. MXCSR.FTZ is ignored. If a source element is denormal relative to input format with MXCSR.DAZ not set, DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

128-bit version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. The upper-bits vector register zeroing behavior of VEX prefix encoding still applies if the destination operand is a *xmm* register. So the upper bits (255:64) of corresponding YMM register are zeroed.

256-bit version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. The upper-bits vector register zeroing behavior of VEX prefix encoding still applies if the destination operand is a *xmm* register. So the upper bits (255:128) of the corresponding YMM register are zeroed.

Note: VEX.vvvv is reserved (must be 1111b).

The diagram below illustrates how data is converted from four packed single precision (in 128 bits) to four half precision (in 64 bits) FP values.

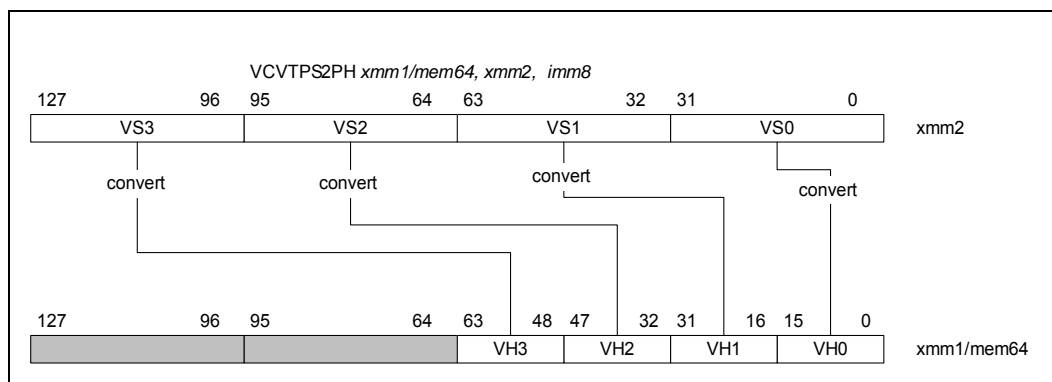


Figure 4-32. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that controls rounding operation. The effect and encoding of RC field are listed in Table 4-17.

Table 4-17. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

Operation

```

vCvt_s2h(SRC1[31:0])
{
  IF Imm[2] = 0
  THEN // using Imm[1:0] for rounding control, see Table 4-17
      RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
  ELSE // using MXCSR.RC for rounding control
      RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
  FI;
}
```

VCVTPS2PH (VEX.256 encoded version)

```

DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[79:64] ← vCvt_s2h(SRC1[159:128]);
DEST[95:80] ← vCvt_s2h(SRC1[191:160]);
DEST[111:96] ← vCvt_s2h(SRC1[223:192]);
DEST[127:112] ← vCvt_s2h(SRC1[255:224]);
DEST[255:128] ← 0; // if DEST is a register
```

VCVTPS2PH (VEX.128 encoded version)

```

DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[VLMAX-1:64] ← 0; // if DEST is a register
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

`__m128i _mm_cvtps_ph (__m128 m1, const int imm);`
`__m128i _mm256_cvtps_ph(__m256 m1, const int imm);`

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

Other Exceptions

Exceptions Type 11 (do not report #AC); additionally
#UD If VEX.W=1.

VERR/VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /4	VERR <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be read.
OF 00 /5	VERW <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be written.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (<i>r</i>)	NA	NA	NA

Description

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

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

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

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

This instruction's operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

Operation

```
IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
  THEN ZF ← 0; FI;
```

Read segment descriptor;

```
IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
```

```
  THEN
```

```
    ZF ← 0;
```

```
  ELSE
```

```
    IF ((Instruction = VERR) and (Segment readable))
```

```
    or ((Instruction = VERW) and (Segment writable))
```

```
      THEN
```

```
        ZF ← 1;
```

```
    FI;
```

```
FI;
```

Flags Affected

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

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode. If the LOCK prefix is used.
-----	--

Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode. If the LOCK prefix is used.
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used.

VEEXTRACTF128 — Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEEXTRACTF128 <i>xmm1/m128, ymm2, imm8</i>	MR	V/V	AVX	Extract 128 bits of packed floating-point values from <i>ymm2</i> and store results in <i>xmm1/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Extracts 128-bits of packed floating-point values from the source operand (second operand) at an 128-bit offset from *imm8[0]* into the destination operand (first operand). The destination may be either an XMM register or an 128-bit memory location.

VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits of the immediate are ignored.

If VEEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEEXTRACTF128 (memory destination form)

CASE (*imm8[0]*) OF
 0: DEST[127:0] ← SRC1[127:0]
 1: DEST[127:0] ← SRC1[255:128]
ESAC.

VEEXTRACTF128 (register destination form)

CASE (*imm8[0]*) OF
 0: DEST[127:0] ← SRC1[127:0]
 1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VEEXTRACTF128: __m128 _mm256_extractf128_ps (__m256 a, int offset);
VEEXTRACTF128: __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEEXTRACTF128: __m128i _mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally
#UD If VEX.L= 0
 If VEX.W=1.

VEEXTRACTI128 — Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEEXTRACTI128 <i>xmm1/m128, ymm2, imm8</i>	RMI	V/V	AVX2	Extract 128 bits of integer data from <i>ymm2</i> and store results in <i>xmm1/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

Extracts 128-bits of packed integer values from the source operand (second operand) at a 128-bit offset from `imm8[0]` into the destination operand (first operand). The destination may be either an XMM register or a 128-bit memory location.

VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits of the immediate are ignored.

An attempt to execute VEEXTRACTI128 encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEEXTRACTI128 (memory destination form)

CASE (imm8[0]) OF

0: DEST[127:0] ← SRC1[127:0]

1: DEST[127:0] ← SRC1[255:128]

ESAC.

VEEXTRACTI128 (register destination form)

CASE (imm8[0]) OF

0: DEST[127:0] ← SRC1[127:0]

1: DEST[127:0] ← SRC1[255:128]

ESAC.

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VEEXTRACTI128: `__m128i _mm256_extracti128_si256(__m256i a, int offset);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD IF VEX.L = 0,
If VEX.W = 1.

VFMADD132PD/VFMADD213PD/VFMADD231PD — Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 98 /r VFMADD132PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 A8 /r VFMADD213PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 B8 /r VFMADD231PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 98 /r VFMADD132PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 A8 /r VFMADD213PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 B8 /r VFMADD231PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add to <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD213PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD231PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD132PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD213PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD231PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132PS/VFMADD213PS/VFMADD231PS — Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 98 /r VFMADD132PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 A8 /r VFMADD213PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 B8 /r VFMADD231PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 98 /r VFMADD132PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 A8 /r VFMADD213PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 B8 /r VFMADD231PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add to <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PS: `__m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);`

VFMADD213PS: `__m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);`

VFMADD231PS: `__m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);`

VFMADD132PS: `__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);`

VFMADD213PS: `__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);`

VFMADD231PS: `__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132SD/VFMADD213SD/VFMADD231SD — Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 99 /r VFMADD132SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 A9 /r VFMADD213SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 B9 /r VFMADD231SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , add to <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-add computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 64-bit memory location and encoded in `rm_field`. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, “+”, “-”, and “*” symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMAADD132SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}[63:0] * \text{SRC3}[63:0] + \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFMAADD213SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[63:0] * \text{DEST}[63:0] + \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFMAADD231SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[63:0] * \text{SRC3}[63:0] + \text{DEST}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
Intel C/C++ Compiler Intrinsic Equivalent

VFMAADD132SD: `__m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);`

VFMAADD213SD: `__m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);`

VFMAADD231SD: `__m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADD132SS/VFMADD213SS/VFMADD231SS — Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 99 /r VFMADD132SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 A9 /r VFMADD213SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 B9 /r VFMADD231SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , add to <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 32-bit memory location and encoded in *rm_field*. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMAADD132SS DEST, SRC2, SRC3

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}[31:0] * \text{SRC3}[31:0] + \text{SRC2}[31:0])$
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VFMAADD213SS DEST, SRC2, SRC3

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[31:0] * \text{DEST}[31:0] + \text{SRC3}[31:0])$
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VFMAADD231SS DEST, SRC2, SRC3

$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[31:0] * \text{SRC3}[63:0] + \text{DEST}[31:0])$
 $\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

VFMAADD132SS: `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);`

VFMAADD213SS: `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);`

VFMAADD231SS: `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD — Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add/subtract elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add/subtract elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADDSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in *reg_field*. The second source operand is a YMM register and encoded in *VEX.vvvv*. The third source operand is a YMM register or a 256-bit memory location and encoded in *rm_field*.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

IF (VEX.128) THEN

DEST[63:0] \leftarrow RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
 DEST[VLMAX-1:128] \leftarrow 0

ELSEIF (VEX.256)

DEST[63:0] \leftarrow RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
 DEST[191:128] \leftarrow RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
 DEST[255:192] \leftarrow RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192])

FI

VFMADDSUB213PD DEST, SRC2, SRC3

IF (VEX.128) THEN

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
 DEST[VLMAX-1:128] \leftarrow 0

ELSEIF (VEX.256)

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
 DEST[191:128] \leftarrow RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
 DEST[255:192] \leftarrow RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192])

FI

VFMADDSUB231PD DEST, SRC2, SRC3

IF (VEX.128) THEN

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
 DEST[VLMAX-1:128] \leftarrow 0

ELSEIF (VEX.256)

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
 DEST[191:128] \leftarrow RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
 DEST[255:192] \leftarrow RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192])

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB213PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB231PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB132PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB213PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB231PD: __m256d __mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS — Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , add/subtract elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , add/subtract elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , add/subtract elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , add/subtract elements in <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADDSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in *reg_field*. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in *rm_field*.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PS: `__m128_mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);`
VFMADDSUB213PS: `__m128_mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);`
VFMADDSUB231PS: `__m128_mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);`
VFMADDSUB132PS: `__m256_mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);`
VFMADDSUB213PS: `__m256_mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);`
VFMADDSUB231PS: `__m256_mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD — Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract/add elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract/add elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (<i>r</i> , <i>w</i>)	VEX.vvvv (<i>r</i>)	ModRM:r/m (<i>r</i>)	NA

Description

VFMSUBADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in *reg_field*. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in *rm_field*.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

IF (VEX.128) THEN

DEST[63:0] \leftarrow RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
 DEST[VLMAX-1:128] \leftarrow 0

ELSEIF (VEX.256)

DEST[63:0] \leftarrow RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
 DEST[191:128] \leftarrow RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
 DEST[255:192] \leftarrow RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192])

FI

VFMSUBADD213PD DEST, SRC2, SRC3

IF (VEX.128) THEN

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
 DEST[VLMAX-1:128] \leftarrow 0

ELSEIF (VEX.256)

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
 DEST[191:128] \leftarrow RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
 DEST[255:192] \leftarrow RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192])

FI

VFMSUBADD231PD DEST, SRC2, SRC3

IF (VEX.128) THEN

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
 DEST[VLMAX-1:128] \leftarrow 0

ELSEIF (VEX.256)

DEST[63:0] \leftarrow RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
 DEST[127:64] \leftarrow RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
 DEST[191:128] \leftarrow RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
 DEST[255:192] \leftarrow RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192])

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD213PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD231PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD132PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD213PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD231PD: __m256d __mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS — Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract/add elements in <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract/add elements in <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract/add elements in <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract/add elements in <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMSUBADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in *reg_field*. The second source operand is a YMM register and encoded in *VEX.vvvv*. The third source operand is a YMM register or a 256-bit memory location and encoded in *rm_field*.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] - SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] - SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] - DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PS: `__m128_mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);`
VFMSUBADD213PS: `__m128_mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);`
VFMSUBADD231PS: `__m128_mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);`
VFMSUBADD132PS: `__m256_mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);`
VFMSUBADD213PS: `__m256_mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);`
VFMSUBADD231PS: `__m256_mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD — Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9A /r VFMSUB132PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 AA /r VFMSUB213PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 BA /r VFMSUB231PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 9A /r VFMSUB132PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 AA /r VFMSUB213PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 BA /r VFMSUB231PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```


Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PD: `__m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);`

VFMSUB213PD: `__m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);`

VFMSUB231PD: `__m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);`

VFMSUB132PD: `__m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);`

VFMSUB213PD: `__m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);`

VFMSUB231PD: `__m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS — Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9A /r VFMSUB132PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 AA /r VFMSUB213PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 BA /r VFMSUB231PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 9A /r VFMSUB132PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 AA /r VFMSUB213PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.0 BA /r VFMSUB231PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , subtract <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 128-bit memory location and encoded in *rm_field*. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PS: `__m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);`

VFMSUB213PS: `__m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);`

VFMSUB231PS: `__m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);`

VFMSUB132PS: `__m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);`

VFMSUB213PS: `__m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);`

VFMSUB231PS: `__m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD — Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9B /r VFMSUB132SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 AB /r VFMSUB213SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 BB /r VFMSUB231SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 64-bit memory location and encoded in *rm_field*. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, “+”, “-”, and “*” symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}[63:0] * \text{SRC3}[63:0] - \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFMSUB213SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[63:0] * \text{DEST}[63:0] - \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFMSUB231SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[63:0] * \text{SRC3}[63:0] - \text{DEST}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SD: `__m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);`

VFMSUB213SD: `__m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);`

VFMSUB231SD: `__m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS — Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9B /r VFMSUB132SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 AB /r VFMSUB213SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 BB /r VFMSUB231SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , subtract <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 32-bit memory location and encoded in *rm_field*. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}[31:0] * \text{SRC3}[31:0] - \text{SRC2}[31:0])$$

$$\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFMSUB213SS DEST, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[31:0] * \text{DEST}[31:0] - \text{SRC3}[31:0])$$

$$\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFMSUB231SS DEST, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(\text{SRC2}[31:0] * \text{SRC3}[63:0] - \text{DEST}[31:0])$$

$$\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SS: `__m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);`

VFMSUB213SS: `__m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);`

VFMSUB231SS: `__m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD — Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9C /r VFNMADD132PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 AC /r VFNMADD213PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 BC /r VFNMADD231PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 9C /r VFNMADD132PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 AC /r VFNMADD213PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 BC /r VFNMADD231PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a

XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
```

IF (VEX.128) THEN
 DEST[VLMAX-1:128] \leftarrow 0
 FI

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PD: `__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);`
 VFMADD213PD: `__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);`
 VFMADD231PD: `__m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);`
 VFMADD132PD: `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);`
 VFMADD213PD: `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);`
 VFMADD231PD: `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS — Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9C /r VFNMADD132PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 AC /r VFNMADD213PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 BC /r VFNMADD231PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 9C /r VFNMADD132PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 AC /r VFNMADD213PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and add to <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.0 BC /r VFNMADD231PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and add to <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in *reg_field*. The second source operand is a YMM register and encoded in *VEX.vvvv*. The third source operand is a YMM register or a 256-bit memory location and encoded in *rm_field*.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD213PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD231PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD132PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD213PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD231PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132SD/VFMADD213SD/VFMADD231SD — Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9D /r VFMADD132SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 AD /r VFMADD213SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 BD /r VFMADD231SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 64-bit memory location and encoded in `rm_field`. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{DEST}[63:0] * \text{SRC3}[63:0]) + \text{SRC2}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFNMADD213SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{SRC2}[63:0] * \text{DEST}[63:0]) + \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFNMADD231SD DEST, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{SRC2}[63:0] * \text{SRC3}[63:0]) + \text{DEST}[63:0])$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
Intel C/C++ Compiler Intrinsic Equivalent

$$\text{VFNMADD132SD: } _m128d_mm_fnmadd_sd(_m128d\ a, _m128d\ b, _m128d\ c);$$

$$\text{VFNMADD213SD: } _m128d_mm_fnmadd_sd(_m128d\ a, _m128d\ b, _m128d\ c);$$

$$\text{VFNMADD231SD: } _m128d_mm_fnmadd_sd(_m128d\ a, _m128d\ b, _m128d\ c);$$
SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADD132SS/VFMADD213SS/VFMADD231SS — Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9D /r VFMADD132SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 AD /r VFMADD213SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and add to <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 BD /r VFMADD231SS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and add to <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in *rm_field*. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, “+”, “-”, and “*” symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{DEST}[31:0] * \text{SRC3}[31:0]) + \text{SRC2}[31:0])$$

$$\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFNMADD213SS DEST, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{SRC2}[31:0] * \text{DEST}[31:0]) + \text{SRC3}[31:0])$$

$$\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
VFNMADD231SS DEST, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{SRC2}[31:0] * \text{SRC3}[63:0]) + \text{DEST}[31:0])$$

$$\text{DEST}[127:32] \leftarrow \text{DEST}[127:32]$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$
Intel C/C++ Compiler Intrinsic Equivalent

$$\text{VFNMADD132SS: } _m128_mm_fnmadd_ss(_m128\ a, _m128\ b, _m128\ c);$$

$$\text{VFNMADD213SS: } _m128_mm_fnmadd_ss(_m128\ a, _m128\ b, _m128\ c);$$

$$\text{VFNMADD231SS: } _m128_mm_fnmadd_ss(_m128\ a, _m128\ b, _m128\ c);$$
SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNSUB132PD/VFNSUB213PD/VFNSUB231PD — Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9E /r VFNSUB132PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 AE /r VFNSUB213PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W1 BE /r VFNSUB231PD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W1 9E /r VFNSUB132PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 AE /r VFNSUB213PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W1 BE /r VFNSUB231PD <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed double-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a

XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
```

IF (VEX.128) THEN
 DEST[VLMAX-1:128] \leftarrow 0
 FI

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB213PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB231PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB132PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB213PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB231PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS — Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9E /r VFNMSUB132PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 AE /r VFNMSUB213PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.128.66.0F38.W0 BE /r VFNMSUB231PS <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .
VEX.DDS.256.66.0F38.W0 9E /r VFNMSUB132PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm1</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.W0 AE /r VFNMSUB213PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm0</i> and <i>ymm1</i> , negate the multiplication result and subtract <i>ymm2/mem</i> and put result in <i>ymm0</i> .
VEX.DDS.256.66.0F38.0 BE /r VFNMSUB231PS <i>ymm0</i> , <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	FMA	Multiply packed single-precision floating-point values from <i>ymm1</i> and <i>ymm2/mem</i> , negate the multiplication result and subtract <i>ymm0</i> and put result in <i>ymm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a

XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL = 4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
```

```

IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VFNMSUB132PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUB213PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUB231PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUB132PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);
VFNMSUB213PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);
VFNMSUB231PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD — Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9F /r VFNMSUB132SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 AF /r VFNMSUB213SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W1 BF /r VFNMSUB231SD <i>xmm0</i> , <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	FMA	Multiply scalar double-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 64-bit memory location and encoded in `rm_field`. The upper bits (`[VLMAX-1:128]`) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, “+”, “-”, and “*” symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{DEST}[63:0] * \text{SRC3}[63:0]) - \text{SRC2}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VFNMSUB213SD DEST, SRC2, SRC3

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{SRC2}[63:0] * \text{DEST}[63:0]) - \text{SRC3}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VFNMSUB231SD DEST, SRC2, SRC3

$\text{DEST}[63:0] \leftarrow \text{RoundFPControl_MXCSR}(-(\text{SRC2}[63:0] * \text{SRC3}[63:0]) - \text{DEST}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64]$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SD: `__m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);`

VFNMSUB213SD: `__m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);`

VFNMSUB231SD: `__m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS — Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9F /r VFNMSUB132SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm1</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 AF /r VFNMSUB213SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm0</i> and <i>xmm1</i> , negate the multiplication result and subtract <i>xmm2/mem</i> and put result in <i>xmm0</i> .
VEX.DDS.LIG.128.66.0F38.W0 BF /r VFNMSUB231SS <i>xmm0, xmm1, xmm2/m32</i>	A	V/V	FMA	Multiply scalar single-precision floating-point value from <i>xmm1</i> and <i>xmm2/mem</i> , negate the multiplication result and subtract <i>xmm0</i> and put result in <i>xmm0</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in *reg_field*. The second source operand is a XMM register and encoded in *VEX.vvvv*. The third source operand is a XMM register or a 32-bit memory location and encoded in *rm_field*. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, “+”, “-”, and “*” symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3

$DEST[31:0] \leftarrow \text{RoundFPControl_MXCSR}(- (DEST[31:0] * SRC3[31:0]) - SRC2[31:0])$

$DEST[127:32] \leftarrow DEST[127:32]$

DEST[VLMAX-1:128] \leftarrow 0

VFNMSUB213SS DEST, SRC2, SRC3

DEST[31:0] \leftarrow RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])

DEST[127:32] \leftarrow DEST[127:32]

DEST[VLMAX-1:128] \leftarrow 0

VFNMSUB231SS DEST, SRC2, SRC3

DEST[31:0] \leftarrow RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) - DEST[31:0])

DEST[127:32] \leftarrow DEST[127:32]

DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SS: `__m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);`

VFNMSUB213SS: `__m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);`

VFNMSUB231SS: `__m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/3 2-bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST \leftarrow SRC1;

BASE_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK \leftarrow SRC3;

VGATHERDPD (VEX.128 version)

FOR j \leftarrow 0 to 1

 i \leftarrow j * 64;

 IF MASK[63+i] THEN

 MASK[i + 63:i] \leftarrow FFFFFFFF_FFFFFFFFH; // extend from most significant bit

 ELSE

 MASK[i + 63:i] \leftarrow 0;

 FI;

ENDFOR

FOR j \leftarrow 0 to 1

 k \leftarrow j * 32;

 i \leftarrow j * 64;

 DATA_ADDR \leftarrow BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;

 IF MASK[63+i] THEN

 DEST[i + 63:i] \leftarrow FETCH_64BITS(DATA_ADDR); // a fault exits the instruction

 FI;

 MASK[i + 63: i] \leftarrow 0;

ENDFOR

```

MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

VGATHERQPD (VEX.128 version)

```

FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
  FI;
  MASK[i + 63: i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

VGATHERQPD (VEX.256 version)

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63: i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

VGATHERDPD (VEX.256 version)

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3

```

```

k ← j * 32;
i ← j * 64;
DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
IF MASK[63:i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);
VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int
scale);
VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);
VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int
scale);
VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);
VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int
scale);
VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);
VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int
scale);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST \leftarrow SRC1;

BASE_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK \leftarrow SRC3;

VGATHERDPS (VEX.128 version)

```

FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  IF MASK[31:i] THEN
    MASK[i + 31:i]  $\leftarrow$  FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i]  $\leftarrow$  0;
  FI;
ENDFOR
MASK[VLMAX-1:128]  $\leftarrow$  0;
FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31:i] THEN
    DEST[i + 31:i]  $\leftarrow$  FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i]  $\leftarrow$  0;
ENDFOR

```

DEST[VLMAX-1:128] \leftarrow 0;

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.128 version)

```

FOR j  $\leftarrow$  0 to 3
  i  $\leftarrow$  j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i]  $\leftarrow$  FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i]  $\leftarrow$  0;
  FI;
ENDFOR
MASK[VLMAX-1:128]  $\leftarrow$  0;
FOR j  $\leftarrow$  0 to 1
  k  $\leftarrow$  j * 64;
  i  $\leftarrow$  j * 32;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i]  $\leftarrow$  FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i]  $\leftarrow$  0;
ENDFOR
MASK[127:64]  $\leftarrow$  0;
DEST[VLMAX-1:64]  $\leftarrow$  0;
(non-masked elements of the mask register have the content of respective element cleared)

```

VGATHERDPS (VEX.256 version)

```

FOR j  $\leftarrow$  0 to 7
  i  $\leftarrow$  j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i]  $\leftarrow$  FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i]  $\leftarrow$  0;
  FI;
ENDFOR
FOR j  $\leftarrow$  0 to 7
  i  $\leftarrow$  j * 32;
  DATA_ADDR  $\leftarrow$  BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i]  $\leftarrow$  FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i]  $\leftarrow$  0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

VGATHERQPS (VEX.256 version)

```

FOR j  $\leftarrow$  0 to 7
  i  $\leftarrow$  j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i]  $\leftarrow$  FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i]  $\leftarrow$  0;
  FI;
ENDFOR

```

```

FOR j ← 0 to 3
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);
VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);
VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);
VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);
VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);
VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);
VGATHERQPS: __m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);
VGATHERQPS: __m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather dword from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST \leftarrow SRC1;

BASE_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK \leftarrow SRC3;

VPGATHERDD (VEX.128 version)

FOR j \leftarrow 0 to 3

 i \leftarrow j * 32;

 IF MASK[31:i] THEN

 MASK[i + 31:i] \leftarrow FFFFFFFFH; // extend from most significant bit

 ELSE

 MASK[i + 31:i] \leftarrow 0;

 FI;

ENDFOR

MASK[VLMAX-1:128] \leftarrow 0;

FOR j \leftarrow 0 to 3

 i \leftarrow j * 32;

 DATA_ADDR \leftarrow BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;

 IF MASK[31:i] THEN

 DEST[i + 31:i] \leftarrow FETCH_32BITS(DATA_ADDR); // a fault exits the instruction

 FI;

 MASK[i + 31:i] \leftarrow 0;

ENDFOR

DEST[VLMAX-1:128] \leftarrow 0;

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQD (VEX.128 version)

```

FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31:i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31:i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

VPGATHERDD (VEX.256 version)

```

FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31:i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31:i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

VPGATHERQD (VEX.256 version)

```

FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31:i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 64;
  i ← j * 32;

```

```

DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
IF MASK[31:i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDD: __m256i _mm256_i32gather_epi32 ( int const * base, __m256i index, const int scale);
VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);
VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST \leftarrow SRC1;

BASE_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK \leftarrow SRC3;

VPGATHERDQ (VEX.128 version)

FOR j \leftarrow 0 to 1

 i \leftarrow j * 64;

 IF MASK[63:i] THEN

 MASK[i + 63:i] \leftarrow FFFFFFFF_FFFFFFFFH; // extend from most significant bit

 ELSE

 MASK[i + 63:i] \leftarrow 0;

 FI;

ENDFOR

FOR j \leftarrow 0 to 1

 k \leftarrow j * 32;

 i \leftarrow j * 64;

 DATA_ADDR \leftarrow BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;

 IF MASK[63:i] THEN

 DEST[i + 63:i] \leftarrow FETCH_64BITS(DATA_ADDR); // a fault exits the instruction

 FI;

 MASK[i + 63:i] \leftarrow 0;

ENDFOR

MASK[VLMAX-1:128] \leftarrow 0;

DEST[VLMAX-1:128] \leftarrow 0;

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.128 version)

```

FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;

```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.256 version)

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR

```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDQ (VEX.256 version)

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 32;
  i ← j * 64;

```

```

DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
IF MASK[63:i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VPGATHERDQ: __m128i _mm_i32gather_epi64 (int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int
scale);
VPGATHERDQ: __m256i _mm256_i32gather_epi64 ( int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, int64 const * base, __m128i index, __m256i mask, const int
scale);
VPGATHERQQ: __m128i _mm_i64gather_epi64 (int64 const * base, __m128i index, const int scale);
VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int
scale);
VPGATHERQQ: __m256i _mm256_i64gather_epi64 (int64 const * base, __m256i index, const int scale);
VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, int64 const * base, __m256i index, __m256i mask, const int
scale);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VINSERTF128 — Insert Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVM	V/V	AVX	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm3/m128</i> into <i>ymm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>ymm1</i> as indicated in <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an insertion of 128-bits of packed floating-point values from the second source operand (third operand) into an the destination operand (first operand) at an 128-bit offset from *imm8*[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

Operation

TEMP[255:0] ← SRC1[255:0]
CASE (*imm8*[0]) OF
 0: TEMP[127:0] ← SRC2[127:0]
 1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTF128: __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
VINSERTF128: __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
VINSERTF128: __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally
#UD If VEX.W = 1.

VINSERTI128 — Insert Packed Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX2	Insert 128-bits of integer data from <i>xmm3/mem</i> and the remaining values from <i>ymm2</i> into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Performs an insertion of 128-bits of packed integer data from the second source operand (third operand) into an the destination operand (first operand) at a 128-bit offset from `imm8[0]`. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

VEX.L must be 1; an attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI128

TEMP[255:0] ← SRC1[255:0]

CASE (imm8[0]) OF

0: TEMP[127:0] ← SRC2[127:0]

1: TEMP[255:128] ← SRC2[127:0]

ESAC

DEST ← TEMP

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTI128: `__m256i _mm256_inserti128_si256 (__m256i a, __m128i b, int offset);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.L = 0,
 If VEX.W = 1.

VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS <i>xmm1</i> , <i>xmm2</i> , <i>m128</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS <i>ymm1</i> , <i>ymm2</i> , <i>m256</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD <i>xmm1</i> , <i>xmm2</i> , <i>m128</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD <i>ymm1</i> , <i>ymm2</i> , <i>m256</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS <i>m128</i> , <i>xmm1</i> , <i>xmm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS <i>m256</i> , <i>ymm1</i> , <i>ymm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD <i>m128</i> , <i>xmm1</i> , <i>xmm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD <i>m256</i> , <i>ymm1</i> , <i>ymm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS - 128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VMASKMOVPS - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VMASKMOVPD - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VMASKMOVPD - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

VMASKMOVPS - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```


VMASKMOVPD - 128-bit store

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]

IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

VMASKMOVPD - 256-bit store

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]

IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]

IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent`__m256 _mm256_maskload_ps(float const *a, __m256i mask)``void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)``__m256d _mm256_maskload_pd(double *a, __m256i mask);``void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);``__m128 _mm128_maskload_ps(float const *a, __m128i mask)``void _mm128_maskstore_ps(float *a, __m128i mask, __m128 b)``__m128d _mm128_maskload_pd(double *a, __m128i mask);``void _mm128_maskstore_pd(double *a, __m128i mask, __m128d b);`**SIMD Floating-Point Exceptions**

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations);

additionally

#UD If VEX.W = 1.

VPBLEND — Blend Packed Dwords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLEND <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX2	Select dwords from <i>xmm2</i> and <i>xmm3/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLEND <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Select dwords from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in <i>imm8</i> and store the values into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

VPBLEND (VEX.256 encoded version)

```

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] ← SRC2[159:128]
ELSE DEST[159:128] ← SRC1[159:128]
IF (imm8[5] == 1) THEN DEST[191:160] ← SRC2[191:160]
ELSE DEST[191:160] ← SRC1[191:160]
IF (imm8[6] == 1) THEN DEST[223:192] ← SRC2[223:192]
ELSE DEST[223:192] ← SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] ← SRC2[255:224]
ELSE DEST[255:224] ← SRC1[255:224]

```

VPBLEND (VEX.128 encoded version)

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
 ELSE DEST[31:0] ← SRC1[31:0]
 IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
 ELSE DEST[63:32] ← SRC1[63:32]
 IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
 ELSE DEST[95:64] ← SRC1[95:64]
 IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
 ELSE DEST[127:96] ← SRC1[127:96]
 DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPBLEND: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)

VPBLEND: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

VPBROADCAST—Broadcast Integer Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB <i>xmm1, xmm2/m8</i>	RM	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 78 /r VPBROADCASTB <i>ymm1, xmm2/m8</i>	RM	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in <i>ymm1</i> .
VEX.128.66.0F38.W0 79 /r VPBROADCASTW <i>xmm1, xmm2/m16</i>	RM	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 79 /r VPBROADCASTW <i>ymm1, xmm2/m16</i>	RM	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in <i>ymm1</i> .
VEX.128.66.0F38.W0 58 /r VPBROADCASTD <i>xmm1, xmm2/m32</i>	RM	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 58 /r VPBROADCASTD <i>ymm1, xmm2/m32</i>	RM	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in <i>ymm1</i> .
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ <i>xmm1, xmm2/m64</i>	RM	V/V	AVX2	Broadcast a qword element in mem to two locations in <i>xmm1</i> .
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ <i>ymm1, xmm2/m64</i>	RM	V/V	AVX2	Broadcast a qword element in mem to four locations in <i>ymm1</i> .
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 <i>ymm1, m128</i>	RM	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Load integer data from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

The destination operand is a YMM register. The source operand is 8-bit, 16-bit 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. VPBROADCASTB/D/W/Q also support XMM register as the source operand.

VBROADCASTI128: The destination operand is a YMM register. The source operand is 128-bit memory location. Register source encodings for VBROADCASTI128 are reserved and will #UD.

VPBROADCASTB/W/D/Q is supported in both 128-bit and 256-bit wide versions.

VBROADCASTI128 is only supported as a 256-bit wide version.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. Attempts to execute any VPBROADCAST* instruction with VEX.W = 1 will cause #UD. If VBROADCASTI128 is encoded with VEX.L = 0, an attempt to execute the instruction encoded with VEX.L = 0 will cause an #UD exception.

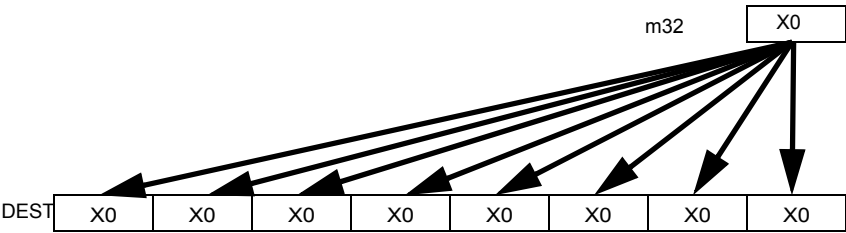


Figure 4-33. VPBROADCASTD Operation (VEX.256 encoded version)

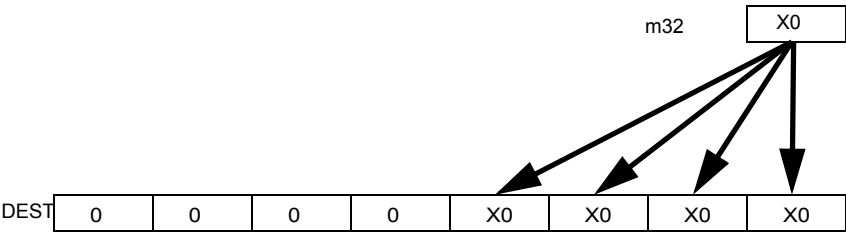


Figure 4-34. VPBROADCASTD Operation (128-bit version)

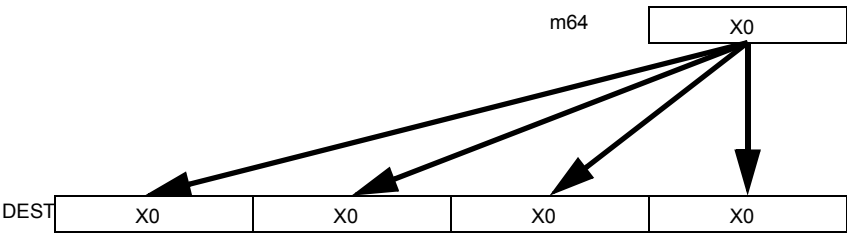


Figure 4-35. VPBROADCASTQ Operation

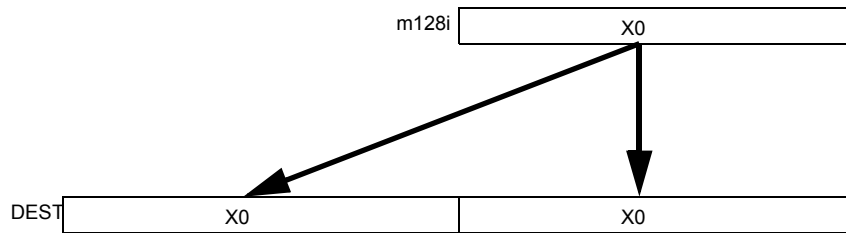


Figure 4-36. VROADCASTI128 Operation

Operation

VPBROADCASTB (VEX.128 encoded version)

```
temp ← SRC[7:0]
FOR j ← 0 TO 15
  DEST[7+j*8:j*8] ← temp
ENDFOR
DEST[VLMAX-1:128] ← 0
```

VPBROADCASTB (VEX.256 encoded version)

```
temp ← SRC[7:0]
FOR j ← 0 TO 31
  DEST[7+j*8:j*8] ← temp
ENDFOR
```

VPBROADCASTW (VEX.128 encoded version)

```
temp ← SRC[15:0]
FOR j ← 0 TO 7
  DEST[15+j*16:j*16] ← temp
ENDFOR
DEST[VLMAX-1:128] ← 0
```

VPBROADCASTW (VEX.256 encoded version)

```
temp ← SRC[15:0]
FOR j ← 0 TO 15
  DEST[15+j*16:j*16] ← temp
ENDFOR
```

VPBROADCASTD (128 bit version)

```
temp ← SRC[31:0]
FOR j ← 0 TO 3
  DEST[31+j*32:j*32] ← temp
ENDFOR
DEST[VLMAX-1:128] ← 0
```

VPBROADCASTD (VEX.256 encoded version)

```
temp ← SRC[31:0]
FOR j ← 0 TO 7
  DEST[31+j*32:j*32] ← temp
ENDFOR
```

VPBROADCASTQ (VEX.128 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[VLMAX-1:128] ← 0
```

VPBROADCASTQ (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
```

VBROADCASTI128

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[VLMAX-1:128] ← temp
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTB:  __m256i _mm256_broadcastb_epi8(__m128i);
VPBROADCASTW:  __m256i _mm256_broadcastw_epi16(__m128i);
VPBROADCASTD:  __m256i _mm256_broadcastd_epi32(__m128i);
VPBROADCASTQ:  __m256i _mm256_broadcastq_epi64(__m128i);
VPBROADCASTB:  __m128i _mm_broadcastb_epi8(__m128i);
VPBROADCASTW:  __m128i _mm_broadcastw_epi16(__m128i);
VPBROADCASTD:  __m128i _mm_broadcastd_epi32(__m128i);
VPBROADCASTQ:  __m128i _mm_broadcastq_epi64(__m128i);
VBROADCASTI128: __m256i _mm256_broadcastsi128_si256(__m128i);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

```
#UD          If VEX.W = 1,
              If VEX.L = 0 for VBROADCASTI128.
```

VPERMD — Full Doublewords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Permute doublewords in <i>ymm3/m256</i> using indexes in <i>ymm2</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Use the index values in each dword element of the first source operand (the second operand) to select a dword element in the second source operand (the third operand), the resultant dword value from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMD encoded with VEX.L = 0 will cause an #UD exception.

Operation

VPERMD (VEX.256 encoded version)

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];

```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMD: `__m256i _mm256_permutevar8x32_epi32(__m256i a, __m256i offsets);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 0 for VPERMD,
 If VEX.W = 1.

VPERMPD — Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Permute double-precision floating-point elements in <i>ymm2/m256</i> using indexes in <i>imm8</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Use two-bit index values in the immediate byte to select a double-precision floating-point element in the source operand; the resultant data from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple location in the destination operand.

An attempt to execute VPERMPD encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (VEX.256 encoded version)

DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD: __m256d _mm256_permute4x64_pd(__m256d a, int control);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 0.

VPERMPS — Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 16 /r VPERMPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Permute single-precision floating-point elements in <i>ymm3/m256</i> using indexes in <i>ymm2</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Use the index values in each dword element of the first source operand (the second operand) to select a single-precision floating-point element in the second source operand (the third operand), the resultant data from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMPS encoded with VEX.L = 0 will cause an #UD exception.

Operation

VPERMPS (VEX.256 encoded version)

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];

```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS: `__m256i _mm256_permutevar8x32_ps(__m256 a, __m256i offsets)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 0,
 If VEX.W = 1.

VPERMQ — Qwords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Permute qwords in <i>ymm2/m256</i> using indexes in <i>imm8</i> and store the result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Use two-bit index values in the immediate byte to select a qword element in the source operand, the resultant qword value from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

An attempt to execute VPERMQ encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMQ (VEX.256 encoded version)
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];

Intel C/C++ Compiler Intrinsic Equivalent

VPERMQ: __m256i _mm256_permute4x64_epi64(__m256i a, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.L = 0.

VPERM2I128 – Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 46 /r ib VPERM2I128 <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVM1	V/V	AVX2	Permute 128-bit integer data in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM1	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

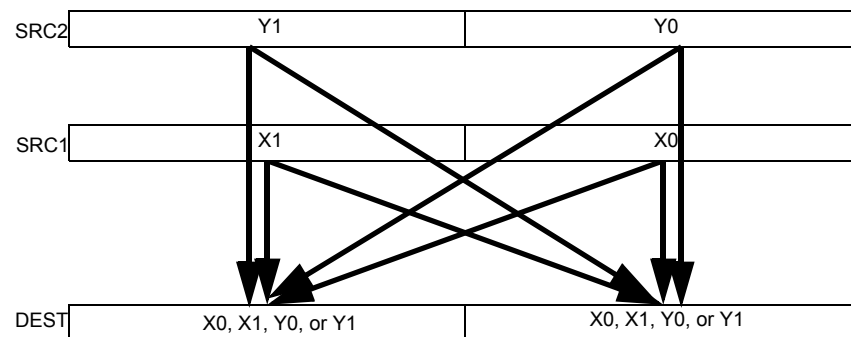


Figure 4-37. VPERM2I128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2I128

CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC
CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally
#UD If VEX.L = 0,
 If VEX.W = 1.

VPERMILPD — Permute Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Permute double-precision floating-point values in <i>xmm2</i> using controls from <i>xmm3/mem</i> and store result in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Permute double-precision floating-point values in <i>ymm2</i> using controls from <i>ymm3/mem</i> and store result in <i>ymm1</i> .
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Permute double-precision floating-point values in <i>xmm2/mem</i> using controls from <i>imm8</i> .
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX	Permute double-precision floating-point values in <i>ymm2/mem</i> using controls from <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Permute double-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of the second source operand (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

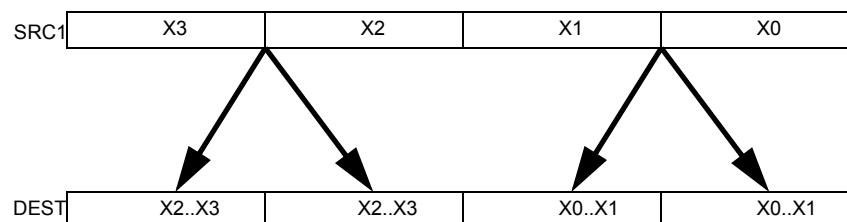


Figure 4-38. VPERMILPD operation

There is one control byte per destination double-precision element. Each control byte is aligned with the low 8 bits of the corresponding double-precision destination element. Each control byte contains a 1-bit select field (see Figure 4-39) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

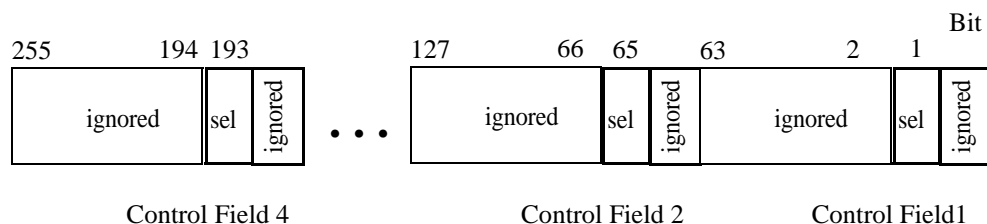


Figure 4-39. VPERMILPD Shuffle Control

(immediate control version)

Permute double-precision floating-point values in the first source operand (second operand) using two, 1-bit control fields in the low 2 bits of the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register.

Note: For the VEX.128.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (256-bit immediate version)

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192] ← SRC1[255:192]

```

VPERMILPD (128-bit immediate version)

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

```

VPERMILPD (256-bit variable version)

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] ← SRC1[255:192]

```

VPERMILPD (128-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
 IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
 IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
 IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
 DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPD: __m128d _mm_permute_pd (__m128d a, int control)
 VPERMILPD: __m256d _mm256_permute_pd (__m256d a, int control)
 VPERMILPD: __m128d _mm_permutevar_pd (__m128d a, __m128i control);
 VPERMILPD: __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.W = 1

VPERMILPS – Permute Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Permute single-precision floating-point values in <i>xmm2</i> using controls from <i>xmm3/mem</i> and store result in <i>xmm1</i> .
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Permute single-precision floating-point values in <i>xmm2/mem</i> using controls from <i>imm8</i> and store result in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Permute single-precision floating-point values in <i>ymm2</i> using controls from <i>ymm3/mem</i> and store result in <i>ymm1</i> .
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX	Permute single-precision floating-point values in <i>ymm2/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

(variable control version)

Permute single-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of corresponding elements the shuffle control (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

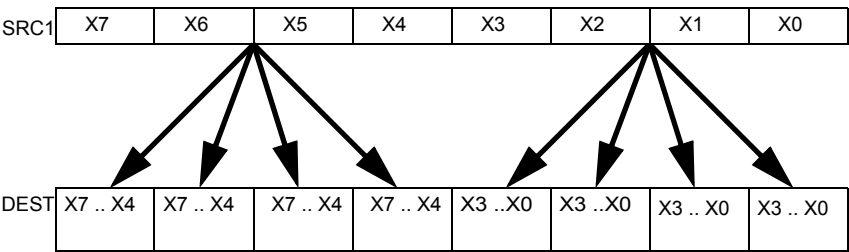


Figure 4-40. VPERMILPS Operation

There is one control byte per destination single-precision element. Each control byte is aligned with the low 8 bits of the corresponding single-precision destination element. Each control byte contains a 2-bit select field (see Figure 4-41) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

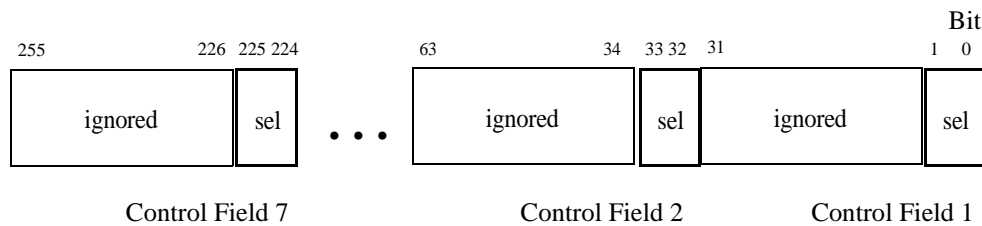


Figure 4-41. VPERMILPS Shuffle Control

(immediate control version)

Permute single-precision floating-point values in the first source operand (second operand) using four 2-bit control fields in the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register. This is similar to a wider version of PSHUFD, just operating on single-precision floating-point values.

Note: For the VEX.128.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

```

Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP ← SRC[31:0];
  1:  TMP ← SRC[63:32];
  2:  TMP ← SRC[95:64];
  3:  TMP ← SRC[127:96];
ESAC;
RETURN TMP
}

```

VPERMILPS (256-bit immediate version)

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);

```

VPERMILPS (128-bit immediate version)

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[VLMAX-1:128] ← 0

```

VPERMILPS (256-bit variable version)

$\text{DEST}[31:0] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[1:0]);$
 $\text{DEST}[63:32] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[33:32]);$
 $\text{DEST}[95:64] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[65:64]);$
 $\text{DEST}[127:96] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[97:96]);$
 $\text{DEST}[159:128] \leftarrow \text{Select4}(\text{SRC1}[255:128], \text{SRC2}[129:128]);$
 $\text{DEST}[191:160] \leftarrow \text{Select4}(\text{SRC1}[255:128], \text{SRC2}[161:160]);$
 $\text{DEST}[223:192] \leftarrow \text{Select4}(\text{SRC1}[255:128], \text{SRC2}[193:192]);$
 $\text{DEST}[255:224] \leftarrow \text{Select4}(\text{SRC1}[255:128], \text{SRC2}[225:224]);$

VPERMILPS (128-bit variable version)

$\text{DEST}[31:0] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[1:0]);$
 $\text{DEST}[63:32] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[33:32]);$
 $\text{DEST}[95:64] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[65:64]);$
 $\text{DEST}[127:96] \leftarrow \text{Select4}(\text{SRC1}[127:0], \text{SRC2}[97:96]);$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS: `__m128 _mm_permute_ps (__m128 a, int control);`
VPERMILPS: `__m256 _mm256_permute_ps (__m256 a, int control);`
VPERMILPS: `__m128 _mm_permutevar_ps (__m128 a, __m128i control);`
VPERMILPS: `__m256 _mm256_permutevar_ps (__m256 a, __m256i control);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.W = 1.

VPERM2F128 — Permute Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVM1	V/V	AVX	Permute 128-bit floating-point fields in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM1	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

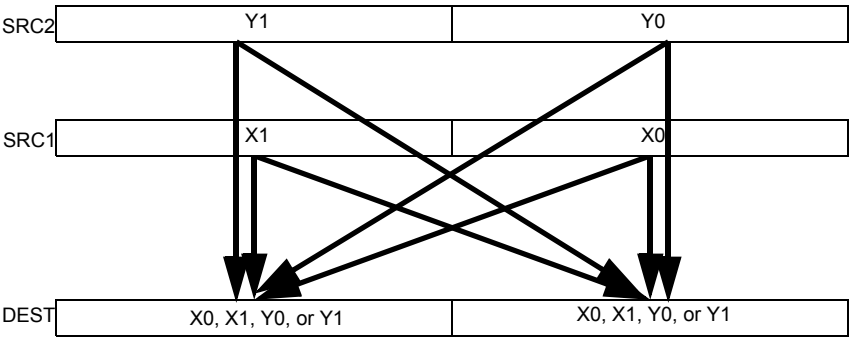


Figure 4-42. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2F128

CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC

IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[VLMAX-1:128] ← 0
FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)
VPERM2F128: __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)
VPERM2F128: __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0
	If VEX.W = 1.

VPMASKMOV – Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOVD <i>xmm1</i> , <i>xmm2</i> , <i>m128</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD <i>ymm1</i> , <i>ymm2</i> , <i>m256</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ <i>xmm1</i> , <i>xmm2</i> , <i>m128</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVQ <i>ymm1</i> , <i>ymm2</i> , <i>m256</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD <i>m128</i> , <i>xmm1</i> , <i>xmm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVD <i>m256</i> , <i>ymm1</i> , <i>ymm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ <i>m128</i> , <i>xmm1</i> , <i>xmm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVQ <i>m256</i> , <i>ymm1</i> , <i>ymm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>ymm2</i> using mask in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv	ModRM:reg (r)	NA

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VPMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VPMASKMOVD - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VPMASKMOVD - 128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VPMASKMOVQ - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VPMASKMOVQ - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VPMASKMOVD - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

VPMASKMOVD - 128-bit store

IF (SRC1[31]) DEST[31:0] \leftarrow SRC2[31:0]
 IF (SRC1[63]) DEST[63:32] \leftarrow SRC2[63:32]
 IF (SRC1[95]) DEST[95:64] \leftarrow SRC2[95:64]
 IF (SRC1[127]) DEST[127:96] \leftarrow SRC2[127:96]

VPMASKMOVQ - 256-bit store

IF (SRC1[63]) DEST[63:0] \leftarrow SRC2[63:0]
 IF (SRC1[127]) DEST[127:64] \leftarrow SRC2[127:64]
 IF (SRC1[191]) DEST[191:128] \leftarrow SRC2[191:128]
 IF (SRC1[255]) DEST[255:192] \leftarrow SRC2[255:192]

VPMASKMOVQ - 128-bit store

IF (SRC1[63]) DEST[63:0] \leftarrow SRC2[63:0]
 IF (SRC1[127]) DEST[127:64] \leftarrow SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

VPMASKMOVD: `__m256i _mm256_maskload_epi32(int const *a, __m256i mask)`
 VPMASKMOVD: `void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)`
 VPMASKMOVQ: `__m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);`
 VPMASKMOVQ: `void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);`
 VPMASKMOVD: `__m128i _mm_maskload_epi32(int const *a, __m128i mask)`
 VPMASKMOVD: `void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)`
 VPMASKMOVQ: `__m128i _mm_maskload_epi64(__int64 const *a, __m128i mask);`
 VPMASKMOVQ: `void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations).

VPSLLVD/VPSLLVQ — Variable Bit Shift Left Logical

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>xmm2</i> left by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>xmm2</i> left by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>ymm2</i> left by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>ymm2</i> left by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

Operation

VPSLLVD (VEX.128 version)

COUNT_0 ← SRC2[31 : 0]

(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)

COUNT_3 ← SRC2[127 : 96];

IF COUNT_0 < 32 THEN

DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);

ELSE

DEST[31:0] ← 0;

(* Repeat shift operation for 2nd through 4th dwords *)

IF COUNT_3 < 32 THEN

DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);

ELSE

DEST[127:96] ← 0;

DEST[VLMAX-1:128] ← 0;

VPSLLVD (VEX.256 version)

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ← 0;

```

VPSLLVQ (VEX.128 version)

```

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;

```

VPSLLVQ (VEX.256 version)

```

COUNT_0 ← SRC2[5 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] ← 0;

```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVD: `__m256i _mm256_sllv_epi32 (__m256i m, __m256i count)`

VPSLLVD: `__m128i _mm_sllv_epi32 (__m128i m, __m128i count)`

VPSLLVQ: `__m256i _mm256_sllv_epi64 (__m256i m, __m256i count)`

VPSLLVQ: `__m128i _mm_sllv_epi64 (__m128i m, __m128i count)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VPSRAVD — Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>xmm2</i> right by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in the sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>ymm2</i> right by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in the sign bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual doubleword data elements in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in each data element are shifted right, the empty high-order bits are filled with the sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31, then the destination data element are filled with the corresponding sign bit of the source element.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

Operation

VPSRAVD (VEX.128 version)

COUNT_0 ← SRC2[31:0]

(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)

COUNT_3 ← SRC2[127:112];

IF COUNT_0 < 32 THEN

DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);

ELSE

For (i = 0 to 31) DEST[i + 0] ← (SRC1[31]);

FI;

(* Repeat shift operation for 2nd through 4th dwords *)

IF COUNT_3 < 32 THEN

DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);

ELSE

For (i = 0 to 31) DEST[i + 96] ← (SRC1[127]);

FI;

DEST[VLMAX-1:128] ← 0;

VPSRAVD (VEX.256 version)

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31]);
FI;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
ELSE
    For (i = 0 to 31) DEST[i + 224] ← (SRC1[255]);
FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD: `__m256i _mm256_srav_epi32 (__m256i m, __m256i count)`

VPSRAVD: `__m128i _mm_srav_epi32 (__m128i m, __m128i count)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

VPSRLVD/VPSRLVQ — Variable Bit Shift Right Logical

Opcode/ Instruction	Op/ EN	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>xmm2</i> right by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>xmm2</i> right by amount specified in the corresponding element of <i>xmm3/m128</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in doublewords in <i>ymm2</i> right by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Shift bits in quadwords in <i>ymm2</i> right by amount specified in the corresponding element of <i>ymm3/m256</i> while shifting in 0s.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

Operation

VPSRLVD (VEX.128 version)

COUNT_0 ← SRC2[31 : 0]

(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)

COUNT_3 ← SRC2[127 : 96];

IF COUNT_0 < 32 THEN

DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);

ELSE

DEST[31:0] ← 0;

(* Repeat shift operation for 2nd through 4th dwords *)

IF COUNT_3 < 32 THEN

DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);

ELSE

DEST[127:96] ← 0;

DEST[VLMAX-1:128] ← 0;

VPSRLVD (VEX.256 version)

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
DEST[255:224] ← 0;

```

VPSRLVQ (VEX.128 version)

```

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
DEST[127:64] ← 0;
DEST[VLMAX-1:128] ← 0;

```

VPSRLVQ (VEX.256 version)

```

COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] ← 0;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VPSRLVD: __m256i _mm256_srlv_epi32 (__m256i m, __m256i count);
VPSRLVD: __m128i _mm_srlv_epi32 (__m128i m, __m128i count);
VPSRLVQ: __m256i _mm256_srlv_epi64 (__m256i m, __m256i count);
VPSRLVQ: __m128i _mm_srlv_epi64 (__m128i m, __m128i count);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.256.66.0F38.W0 0E /r VTESTPS <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.128.66.0F38.W0 0F /r VTESTPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.
VEX.256.66.0F38.W0 0F /r VTESTPD <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VTESTPS (128-bit version)

TEMP[127:0] \leftarrow SRC[127:0] AND DEST[127:0]
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
 THEN ZF \leftarrow 1;
 ELSE ZF \leftarrow 0;

TEMP[127:0] \leftarrow SRC[127:0] AND NOT DEST[127:0]
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
 THEN CF \leftarrow 1;
 ELSE CF \leftarrow 0;
 DEST (unmodified)
 AF \leftarrow OF \leftarrow PF \leftarrow SF \leftarrow 0;

VTESTPS (VEX.256 encoded version)

TEMP[255:0] \leftarrow SRC[255:0] AND DEST[255:0]
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] = TEMP[255] = 0)
 THEN ZF \leftarrow 1;
 ELSE ZF \leftarrow 0;

TEMP[255:0] \leftarrow SRC[255:0] AND NOT DEST[255:0]
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] = TEMP[255] = 0)
 THEN CF \leftarrow 1;
 ELSE CF \leftarrow 0;
 DEST (unmodified)
 AF \leftarrow OF \leftarrow PF \leftarrow SF \leftarrow 0;

VTESTPD (128-bit version)

TEMP[127:0] \leftarrow SRC[127:0] AND DEST[127:0]
 IF (TEMP[63] = TEMP[127] = 0)
 THEN ZF \leftarrow 1;
 ELSE ZF \leftarrow 0;

TEMP[127:0] \leftarrow SRC[127:0] AND NOT DEST[127:0]
 IF (TEMP[63] = TEMP[127] = 0)
 THEN CF \leftarrow 1;
 ELSE CF \leftarrow 0;
 DEST (unmodified)
 AF \leftarrow OF \leftarrow PF \leftarrow SF \leftarrow 0;

VTESTPD (VEX.256 encoded version)

TEMP[255:0] \leftarrow SRC[255:0] AND DEST[255:0]
 IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
 THEN ZF \leftarrow 1;
 ELSE ZF \leftarrow 0;

TEMP[255:0] \leftarrow SRC[255:0] AND NOT DEST[255:0]
 IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
 THEN CF \leftarrow 1;
 ELSE CF \leftarrow 0;
 DEST (unmodified)
 AF \leftarrow OF \leftarrow PF \leftarrow SF \leftarrow 0;

Intel C/C++ Compiler Intrinsic Equivalent

VTESTPS

```
int _mm256_testz_ps (__m256 s1, __m256 s2);
int _mm256_testc_ps (__m256 s1, __m256 s2);
int _mm256_testnzc_ps (__m256 s1, __m128 s2);
int _mm_testz_ps (__m128 s1, __m128 s2);
int _mm_testc_ps (__m128 s1, __m128 s2);
int _mm_testnzc_ps (__m128 s1, __m128 s2);
```

VTESTPD

```
int _mm256_testz_pd (__m256d s1, __m256d s2);
int _mm256_testc_pd (__m256d s1, __m256d s2);
int _mm256_testnzc_pd (__m256d s1, __m256d s2);
int _mm_testz_pd (__m128d s1, __m128d s2);
int _mm_testc_pd (__m128d s1, __m128d s2);
int _mm_testnzc_pd (__m128d s1, __m128d s2);
```

Flags Affected

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

VZEROALL—Zero All YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.OF.WIG 77 VZEROALL	NP	V/V	AVX	Zero all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROALL (VEX.256 encoded version)

IF (64-bit mode)

```

YMM0[VLMAX-1:0] ← 0
YMM1[VLMAX-1:0] ← 0
YMM2[VLMAX-1:0] ← 0
YMM3[VLMAX-1:0] ← 0
YMM4[VLMAX-1:0] ← 0
YMM5[VLMAX-1:0] ← 0
YMM6[VLMAX-1:0] ← 0
YMM7[VLMAX-1:0] ← 0
YMM8[VLMAX-1:0] ← 0
YMM9[VLMAX-1:0] ← 0
YMM10[VLMAX-1:0] ← 0
YMM11[VLMAX-1:0] ← 0
YMM12[VLMAX-1:0] ← 0
YMM13[VLMAX-1:0] ← 0
YMM14[VLMAX-1:0] ← 0
YMM15[VLMAX-1:0] ← 0

```

ELSE

```

YMM0[VLMAX-1:0] ← 0
YMM1[VLMAX-1:0] ← 0
YMM2[VLMAX-1:0] ← 0
YMM3[VLMAX-1:0] ← 0
YMM4[VLMAX-1:0] ← 0
YMM5[VLMAX-1:0] ← 0
YMM6[VLMAX-1:0] ← 0
YMM7[VLMAX-1:0] ← 0
YMM8-15: Unmodified

```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL: `_mm256_zeroall()`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

VZEROUPPER—Zero Upper Bits of YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	NP	V/V	AVX	Zero upper 128 bits of all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROUPPER

IF (64-bit mode)

```

YMM0[VLMAX-1:128] ← 0
YMM1[VLMAX-1:128] ← 0
YMM2[VLMAX-1:128] ← 0
YMM3[VLMAX-1:128] ← 0
YMM4[VLMAX-1:128] ← 0
YMM5[VLMAX-1:128] ← 0
YMM6[VLMAX-1:128] ← 0
YMM7[VLMAX-1:128] ← 0
YMM8[VLMAX-1:128] ← 0
YMM9[VLMAX-1:128] ← 0
YMM10[VLMAX-1:128] ← 0
YMM11[VLMAX-1:128] ← 0
YMM12[VLMAX-1:128] ← 0
YMM13[VLMAX-1:128] ← 0
YMM14[VLMAX-1:128] ← 0
YMM15[VLMAX-1:128] ← 0

```

ELSE

```

YMM0[VLMAX-1:128] ← 0
YMM1[VLMAX-1:128] ← 0
YMM2[VLMAX-1:128] ← 0
YMM3[VLMAX-1:128] ← 0
YMM4[VLMAX-1:128] ← 0
YMM5[VLMAX-1:128] ← 0
YMM6[VLMAX-1:128] ← 0
YMM7[VLMAX-1:128] ← 0
YMM8-15: unmodified

```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER: `_mm256_zeroupper()`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

WAIT/FWAIT—Wait

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9B	WAIT	NP	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	NP	Valid	Valid	Check pending unmasked floating-point exceptions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

CheckForPendingUnmaskedFloatingPointExceptions;

FPU Flags Affected

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

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 09	WBINVD	NP	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
 #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 OF AE /2 WRFSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
REX.W + F3 OF AE /2 WRFSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 OF AE /3 WRGSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
REX.W + F3 OF AE /3 WRGSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

Operation

FS/GS segment base address ← SRC;

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

```
WRFSBASE:    void _writefsbase_u32( unsigned int );
WRFSBASE:    _writefsbase_u64( unsigned __int64 );
WRGSBASE:    void _writegsbase_u32( unsigned int );
WRGSBASE:    _writegsbase_u64( unsigned __int64 );
```

Protected Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.FSGSBASE[bit 16] = 0.
	If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0
#GP(0)	If the source register contains a non-canonical address.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 30	WRMSR	NP	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 35, “Model-Specific Registers (MSRs)”, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

MSR[ECX] ← EDX:EAX;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the value in ECX specifies a reserved or unimplemented MSR address.</p> <p>If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.</p> <p>If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.</p>
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	<p>If the value in ECX specifies a reserved or unimplemented MSR address.</p> <p>If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.</p> <p>If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.</p>
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.
--------	---

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRPKRU—Write Data to User Page Key Register

Opcode*	Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
OF 01 EF	WRPKRU	NP	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The WRPKRU instruction loads the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, a general-protection exception (#GP) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

In 64-bit mode, WRPKRU ignores bits 63:32 of each of RAX, RCX, and RDX.

Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If ECX ≠ 0.
 If EDX ≠ 0.
 #UD If the LOCK prefix is used.
 If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE ¹	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an "XRELEASE-enabled" instruction to end lock elision on the instruction memory operand address.

NOTES:

- Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 15.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REP NZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,

- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE

IF XACQUIRE-enabled instruction

THEN

IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN

HLE_NEST_COUNT++

IF (HLE_NEST_COUNT = 1) THEN

HLE_ACTIVE ← 1

IF 64-bit mode

THEN

restartRIP ← instruction pointer of the XACQUIRE-enabled instruction

ELSE

restartEIP ← instruction pointer of the XACQUIRE-enabled instruction

FI;

Enter HLE Execution (* record register state, start tracking memory state *)

FI; (* HLE_NEST_COUNT = 1 *)

IF ElisionBufferAvailable

THEN

Allocate elision buffer

Record address and data for forwarding and commit checking

Perform elision

ELSE

Perform lock acquire operation transactionally but without elision

FI;

ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)

GOTO HLE_ABORT_PROCESSING

FI;

ELSE

Treat instruction as non-XACQUIRE F2H prefixed legacy instruction

FI;

XRELEASE

IF XRELEASE-enabled instruction

THEN

IF (HLE_NEST_COUNT > 0)

THEN

HLE_NEST_COUNT--

IF lock address matches in elision buffer THEN

IF lock satisfies address and value requirements THEN

Deallocate elision buffer

ELSE

GOTO HLE_ABORT_PROCESSING

FI;

FI;

IF (HLE_NEST_COUNT = 0)

THEN

IF NoAllocatedElisionBuffer

THEN

```

        Try to commit transactional execution
        IF fail to commit transactional execution
            THEN
                GOTO HLE_ABORT_PROCESSING;
            ELSE (* commit success *)
                HLE_ACTIVE ← 0
        FI;
    ELSE
        GOTO HLE_ABORT_PROCESSING
    FI;
FI;
FI; (* HLE_NEST_COUNT > 0 *)
ELSE
    Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE ← 0
    HLE_NEST_COUNT ← 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP ← restartRIP
        ELSE
            EIP ← restartEIP
    FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END

```

SIMD Floating-Point Exceptions

None

Other Exceptions

#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.

XABORT — Transactional Abort

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

Operation

XABORT

```
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;
```

(* For any RTM abort condition encountered during RTM execution *)

```
RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XABORT: void _xabort(unsigned int);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD CUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
If LOCK prefix is used.

XADD—Exchange and Add

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C0 /r	XADD <i>r/m8, r8</i>	MR	Valid	Valid	Exchange <i>r8</i> and <i>r/m8</i> ; load sum into <i>r/m8</i> .
REX + OF C0 /r	XADD <i>r/m8*, r8*</i>	MR	Valid	N.E.	Exchange <i>r8</i> and <i>r/m8</i> ; load sum into <i>r/m8</i> .
OF C1 /r	XADD <i>r/m16, r16</i>	MR	Valid	Valid	Exchange <i>r16</i> and <i>r/m16</i> ; load sum into <i>r/m16</i> .
OF C1 /r	XADD <i>r/m32, r32</i>	MR	Valid	Valid	Exchange <i>r32</i> and <i>r/m32</i> ; load sum into <i>r/m32</i> .
REX.W + OF C1 /r	XADD <i>r/m64, r64</i>	MR	Valid	N.E.	Exchange <i>r64</i> and <i>r/m64</i> ; load sum into <i>r/m64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (<i>r, w</i>)	ModRM:reg (<i>w</i>)	NA	NA

Description

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

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

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XBEGIN — Transactional Begin

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

Operation

XBEGIN

```

IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
  THEN
    RTM_NEST_COUNT++
    IF RTM_NEST_COUNT = 1 THEN
      IF 64-bit Mode
        THEN
          fallbackRIP ← RIP + SignExtend64(IMM)
                                (* RIP is instruction following XBEGIN instruction *)
        ELSE
          fallbackEIP ← EIP + SignExtend32(IMM)
                                (* EIP is instruction following XBEGIN instruction *)
      FI;

      IF (64-bit mode)
        THEN IF (fallbackRIP is not canonical)
          THEN #GP(0)
        FI;
      ELSE IF (fallbackEIP outside code segment limit)
        THEN #GP(0)
      FI;

      RTM_ACTIVE ← 1
      Enter RTM Execution (* record register state, start tracking memory state*)

```

```

    FI; (* RTM_NEST_COUNT = 1 *)
ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
    GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END

```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN: unsigned int _xbegin(void);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
If LOCK prefix is used.

#GP(0) If the fallback address is outside the CS segment.

Real-Address Mode Exceptions

#GP(0) If the fallback address is outside the address space 0000H and FFFFH.

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
If LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the fallback address is outside the address space 0000H and FFFFH.

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
If LOCK prefix is used.

#GP(0) If the fallback address is non-canonical.

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
90+ <i>rw</i>	XCHG AX, <i>r16</i>	0	Valid	Valid	Exchange <i>r16</i> with AX.
90+ <i>rw</i>	XCHG <i>r16</i> , AX	0	Valid	Valid	Exchange AX with <i>r16</i> .
90+ <i>rd</i>	XCHG EAX, <i>r32</i>	0	Valid	Valid	Exchange <i>r32</i> with EAX.
REX.W + 90+ <i>rd</i>	XCHG RAX, <i>r64</i>	0	Valid	N.E.	Exchange <i>r64</i> with RAX.
90+ <i>rd</i>	XCHG <i>r32</i> , EAX	0	Valid	Valid	Exchange EAX with <i>r32</i> .
REX.W + 90+ <i>rd</i>	XCHG <i>r64</i> , RAX	0	Valid	N.E.	Exchange RAX with <i>r64</i> .
86 / <i>r</i>	XCHG <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
REX + 86 / <i>r</i>	XCHG <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
86 / <i>r</i>	XCHG <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
REX + 86 / <i>r</i>	XCHG <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
87 / <i>r</i>	XCHG <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Exchange <i>r16</i> with word from <i>r/m16</i> .
87 / <i>r</i>	XCHG <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Exchange word from <i>r/m16</i> with <i>r16</i> .
87 / <i>r</i>	XCHG <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Exchange <i>r32</i> with doubleword from <i>r/m32</i> .
REX.W + 87 / <i>r</i>	XCHG <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Exchange <i>r64</i> with quadword from <i>r/m64</i> .
87 / <i>r</i>	XCHG <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Exchange doubleword from <i>r/m32</i> with <i>r32</i> .
REX.W + 87 / <i>r</i>	XCHG <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Exchange quadword from <i>r/m64</i> with <i>r64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
0	AX/EAX/RAX (<i>r</i> , <i>w</i>)	opcode + <i>rd</i> (<i>r</i> , <i>w</i>)	NA	NA
0	opcode + <i>rd</i> (<i>r</i> , <i>w</i>)	AX/EAX/RAX (<i>r</i> , <i>w</i>)	NA	NA
MR	ModRM: <i>r/m</i> (<i>r</i> , <i>w</i>)	ModRM:reg (<i>r</i>)	NA	NA
RM	ModRM:reg (<i>w</i>)	ModRM: <i>r/m</i> (<i>r</i>)	NA	NA

Description

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

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on bus locking.)

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

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

TEMP \leftarrow DEST;
 DEST \leftarrow SRC;
 SRC \leftarrow TEMP;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>If either operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XEND — Transactional End

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
OF 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
```

```
    SIGNAL #GP
```

```
ELSE
```

```
    RTM_NEST_COUNT--
```

```
    IF (RTM_NEST_COUNT = 0) THEN
```

```
        Try to commit transaction
```

```
        IF fail to commit transactional execution
```

```
            THEN
```

```
                GOTO RTM_ABORT_PROCESSING;
```

```
            ELSE (* commit success *)
```

```
                RTM_ACTIVE ← 0
```

```
        FI;
```

```
    FI;
```

```
FI;
```

(* For any RTM abort condition encountered during RTM execution *)

```
RTM_ABORT_PROCESSING:
```

```
    Restore architectural register state
```

```
    Discard memory updates performed in transaction
```

```
    Update EAX with status
```

```
    RTM_NEST_COUNT ← 0
```

```
    RTM_ACTIVE ← 0
```

```
    IF 64-bit Mode
```

```
        THEN
```

```
            RIP ← fallbackRIP
```

```
        ELSE
```

```
            EIP ← fallbackEIP
```

```
    FI;
```

```
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XEND: void _xend(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

- #UD CUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
 If LOCK or 66H or F2H or F3H prefix is used.
- #GP(0) If RTM_ACTIVE = 0.

XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 D0	XGETBV	NP	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

Operation

EDX:EAX ← XCR[ECX];

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XGETBV: unsigned __int64 _xgetbv(unsigned int);

Protected Mode Exceptions

- #GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
 If CR4.OSXSAVE[bit 18] = 0.
 If the LOCK prefix is used.
 If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions

- #GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
 If CR4.OSXSAVE[bit 18] = 0.
 If the LOCK prefix is used.
 If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT <i>m8</i>	NP	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	NP	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	NP	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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

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

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

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
  THEN
    AL ← (DS:BX + ZeroExtend(AL));
  ELSE IF (AddressSize = 32)
    AL ← (DS:EBX + ZeroExtend(AL)); FI;
  ELSE (AddressSize = 64)
    AL ← (RBX + ZeroExtend(AL));
  FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) 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.

#UD If the LOCK prefix is used.

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.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

XOR—Logical Exclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
34 <i>ib</i>	XOR AL, <i>imm8</i>	I	Valid	Valid	AL XOR <i>imm8</i> .
35 <i>iw</i>	XOR AX, <i>imm16</i>	I	Valid	Valid	AX XOR <i>imm16</i> .
35 <i>id</i>	XOR EAX, <i>imm32</i>	I	Valid	Valid	EAX XOR <i>imm32</i> .
REX.W + 35 <i>id</i>	XOR RAX, <i>imm32</i>	I	Valid	N.E.	RAX XOR <i>imm32</i> (<i>sign-extended</i>).
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> XOR <i>imm8</i> .
REX + 80 /6 <i>ib</i>	XOR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> XOR <i>imm8</i> .
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm16</i> .
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm32</i> .
REX.W + 81 /6 <i>id</i>	XOR <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm32</i> (<i>sign-extended</i>).
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm8</i> (<i>sign-extended</i>).
83 /6 <i>ib</i>	XOR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm8</i> (<i>sign-extended</i>).
REX.W + 83 /6 <i>ib</i>	XOR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm8</i> (<i>sign-extended</i>).
30 /r	XOR <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> XOR <i>r8</i> .
REX + 30 /r	XOR <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m8</i> XOR <i>r8</i> .
31 /r	XOR <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> XOR <i>r16</i> .
31 /r	XOR <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> XOR <i>r32</i> .
REX.W + 31 /r	XOR <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> XOR <i>r64</i> .
32 /r	XOR <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> XOR <i>r/m8</i> .
REX + 32 /r	XOR <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r8</i> XOR <i>r/m8</i> .
33 /r	XOR <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> XOR <i>r/m16</i> .
33 /r	XOR <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> XOR <i>r/m32</i> .
REX.W + 33 /r	XOR <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> XOR <i>r/m64</i> .

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

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

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

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← DEST XOR SRC;

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57 /r XORPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 57 /r VXORPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 57 /r VXORPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical exclusive-OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

XORPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VXORPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VXORPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

XORPD: `__m128d _mm_xor_pd(__m128d a, __m128d b)`

VXORPD: `__m256d _mm256_xor_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 57 /r XORPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.OF.WIG 57 /r VXORPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.OF.WIG 57 /r VXORPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical exclusive-OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

XORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VXORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VXORPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE XOR } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE XOR } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE XOR } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE XOR } \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE XOR } \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE XOR } \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE XOR } \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE XOR } \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

XORPS: `__m128 _mm_xor_ps(__m128 a, __m128 b)`
 VXORPS: `__m256 _mm256_xor_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AE /5	XRSTOR <i>mem</i>	M	Valid	Valid	Restore state components specified by EDX:EAX from <i>mem</i> .
REX.W+ OF AE /5	XRSTOR64 <i>mem</i>	M	Valid	N.E.	Restore state components specified by EDX:EAX from <i>mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.8, “Operation of XRSTOR,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If $RFBM[i] = 0$, XRSTOR does not update state component i .¹
- If $RFBM[i] = 1$ and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component i .
- If $RFBM[i] = 1$ and $XSTATE_BV[i] = 1$, XRSTOR loads state component i from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component i for which $RFBM[i] = 1$ and $XSTATE_BV[i] = 1$; it tracks as modified any state component i for which $RFBM[i] = 0$.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```

RFBM ← XCR0 AND EDX:EAX; /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
IF in VMX non-root operation
    THEN VMXNR ← 1;

```

1. There is an exception if $RFBM[1] = 0$ and $RFBM[2] = 1$. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

```

    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        If RFBM[0] = 1
            THEN
                IF RSTORMASK[0] = 1
                    THEN load x87 state from legacy region of XSAVE area;
                ELSE initialize x87 state;
            FI;
        FI;
        If RFBM[1] = 1
            THEN
                IF RSTORMASK[1] = 1
                    THEN load XMM registers from legacy region of XSAVE area;
                ELSE set all XMM registers to 0;
            FI;
        FI;
        If RFBM[2] = 1
            THEN
                IF RSTORMASK[2] = 1
                    THEN load AVX state from extended region (standard format) of XSAVE area;
                ELSE initialize AVX state;
            FI;
        FI;
        If RFBM[1] = 1 or RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;
    FI;
ELSE
    /* Compacted form of XRSTOR */
    IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
        THEN /* compacted form not supported */
            #GP(0);
        FI;
        If RFBM[0] = 1
            THEN
                IF RSTORMASK[0] = 1
                    THEN load x87 state from legacy region of XSAVE area;
                ELSE initialize x87 state;
            FI;
        FI;
        If RFBM[1] = 1
            THEN
                IF RSTORMASK[1] = 1
                    THEN load SSE state from legacy region of XSAVE area;
                ELSE initialize SSE state;
            FI;
        FI;
        If RFBM[2] = 1
            THEN

```

```

        IF RSTORMASK[2] = 1
            THEN load AVX state from extended region (compacted format) of XSAVE area;
            ELSE initialize AVX state;
        FI;
    FI;
XRSTOR_INFO ← (CPL, VMXNR, LAXA, COMPMASK);

```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```

XRSTOR:    void _xrstor( void *, unsigned __int64);
XRSTOR:    void _xrstor64( void *, unsigned __int64);

```

Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If any of the LOCK, 66H, F3H or F2H prefixes is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p>
-----	--

If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.

If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.

If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If a memory address is in a non-canonical form.

If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.

If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.

If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C7 /3	XRSTORS <i>mem</i>	M	Valid	Valid	Restore state components specified by EDX:EAX from <i>mem</i> .
REX.W+ OF C7 /3	XRSTORS64 <i>mem</i>	M	Valid	N.E.	Restore state components specified by EDX:EAX from <i>mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.12, “Operation of XRSTORS,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area”).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```

RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
```

LAXA ← linear address of XSAVE area;

```

If RFBM[0] = 1
    THEN
        IF RSTORMASK[0] = 1
            THEN load x87 state from legacy region of XSAVE area;
            ELSE initialize x87 state;
        FI;
    FI;
If RFBM[1] = 1
    THEN
        IF RSTORMASK[1] = 1
            THEN load SSE state from legacy region of XSAVE area;
            ELSE initialize SSE state;
        FI;
    FI;
If RFBM[2] = 1
    THEN
        IF RSTORMASK[2] = 1
            THEN load AVX state from extended region (compact format) of XSAVE area;
            ELSE initialize AVX state;
        FI;
    FI;
XRSTOR_INFO ← (CPL, VMXNR, LAXA, COMPMASK);

```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```

XRSTORS:   void _xrstors( void *, unsigned __int64);
XRSTORS64: void _xrstors64( void *, unsigned __int64);

```

Protected Mode Exceptions

#GP(0)	<p>If CPL > 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If any of the LOCK, 66H, F3H or F2H prefixes is used.</p>
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check

exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If any of the LOCK, 66H, F3H or F2H prefixes is used.</p>

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	<p>If CPL > 0.</p> <p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If any of the LOCK, 66H, F3H or F2H prefixes is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protec-</p>

tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVE—Save Processor Extended States

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AE /4	XSAVE <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> .
REX.W+ OF AE /4	XSAVE64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.7, “Operation of XSAVE,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if $RFBM[i] = 1$.¹
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If $RFBM[i] = 1$, XSAVE writes $XSTATE_BV[i]$ with the value of $XINUSE[i]$. ($XINUSE$ is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) If $RFBM[i] = 0$, XSAVE writes $XSTATE_BV[i]$ with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

$RFBM \leftarrow XCR0 \text{ AND } EDX:EAX$; /* bitwise logical AND */

$OLD_BV \leftarrow XSTATE_BV$ field from XSAVE header;

IF $RFBM[0] = 1$

THEN store x87 state into legacy region of XSAVE area;

FI;

IF $RFBM[1] = 1$

THEN store XMM registers into legacy region of XSAVE area;

FI;

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either $RFBM[1]$ or $RFBM[2]$ is 1.

IF RFBM[2] = 1

THEN store AVX state into extended region of XSAVE area;

FI;

IF RFBM[1] = 1 or RFBM[2] = 1

THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;

FI;

XSTATE_BV field in XSAVE header \leftarrow (OLD_BV AND \sim RFBM) OR (XINUSE AND RFBM);

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVE: void _xsave(void *, unsigned __int64);

XSAVE: void _xsave64(void *, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States with Compaction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C7 /4	XSAVEC <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> with compaction.
REX.W+ OF C7 /4	XSAVEC64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> with compaction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.10, “Operation of XSAVEC,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if $RFBM[i] = 1$ and $XINUSE[i] = 1$.¹ (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.^{2,3} (See Section 13.4.2, “XSAVE Header.”) XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to $RFBM[62:0]$. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

$RFBM \leftarrow XCR0 \text{ AND } EDX:EAX$; /* bitwise logical AND */
 $COMPMASK \leftarrow RFBM \text{ OR } 80000000_00000000H$;

IF $RFBM[0] = 1$ and $XINUSE[0] = 1$

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but $XINUSE[1]$ may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as $RFBM[1] = 1$.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but $XINUSE[1]$ may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets $XSTATE_BV[1]$ to 1 as long as $RFBM[1] = 1$.

```

    THEN store x87 state into legacy region of XSAVE area;
FI;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
FI;

XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;1
XCOMP_BV field in XSAVE header ← COMPMASK;

```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVEC:    void _xsavc( void *, unsigned __int64);
XSAVEC64:  void _xsavc64( void *, unsigned __int64);

```

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

1. If MXCSR does not have its initial value of 1F80H, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1, regardless of the value of XINUSE[1].

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	<p>If the memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If any of the LOCK, 66H, F3H or F2H prefixes is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF AE /6 XSAVEOPT <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.
REX.W + OF AE /6 XSAVEOPT64 <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.9, “Operation of XSAVEOPT,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it uses compaction and that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if $RFBM[i] = 1$ and $XINUSE[i] = 1$.¹ (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVEOPT corresponds to that last execution of XRTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If $RFBM[i] = 1$, XSAVEOPT writes $XSTATE_BV[i]$ with the value of $XINUSE[i]$. If $RFBM[i] = 0$, XSAVEOPT writes $XSTATE_BV[i]$ with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

$RFBM \leftarrow XCR0 \text{ AND } EDX:EAX$; /* bitwise logical AND */

$OLD_BV \leftarrow XSTATE_BV$ field from XSAVE header;

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if $RFBM[1] = 1$ or $RFBM[2] = 1$, regardless of the value of XINUSE.

IF in VMX non-root operation

THEN VMXNR \leftarrow 1;

ELSE VMXNR \leftarrow 0;

FI;

LAXA \leftarrow linear address of XSAVE area;

COMPMASK \leftarrow 00000000_00000000H;

IF XRSTOR_INFO = \langle CPL, VMXNR, LAXA, COMPMASK \rangle

THEN MODOPT \leftarrow 1;

ELSE MODOPT \leftarrow 0;

FI;

IF RFBM[0] = 1 and XINUSE[0] = 1

THEN store x87 state into legacy region of XSAVE area;

/* might avoid saving if x87 state is not modified and MODOPT = 1 */

FI;

IF RFBM[1] = 1 and XINUSE[1]

THEN store XMM registers into legacy region of XSAVE area;

/* might avoid saving if XMM registers are not modified and MODOPT = 1 */

FI;

IF RFBM[2] = 1 AND XINUSE[2] = 1

THEN store AVX state into extended region of XSAVE area;

/* might avoid saving if AVX state is not modified and MODOPT = 1 */

FI;

IF RFBM[1] = 1 or RFBM[2] = 1

THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;

FI;

XSTATE_BV field in XSAVE header \leftarrow (OLD_BV AND \sim RFBM) OR (XINUSE AND RFBM);

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEOPT: void _xsaveopt(void *, unsigned __int64);

XSAVEOPT: void _xsaveopt64(void *, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used. If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
-----	--

#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used. If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used. If 66H, F3H or F2H prefix is used.

XSAVES—Save Processor Extended States Supervisor

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C7 /5	XSAVES <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.
REX.W+ OF C7 /5	XSAVES64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.11, “Operation of XSAVES,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.¹ (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.² (See Section 13.4.2, “XSAVE Header.”) XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

-
1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
 2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```

RFBM ← XCRO AND EDX:EAX; /* bitwise logical AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
IF XRSTOR_INFO = ⟨CPL, VMXNR, LAXA, COMPMASK⟩
    THEN MODOPT ← 1;
    ELSE MODOPT ← 0;
FI;

IF RFBM[0] = 1 and XINUSE[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
    /* might avoid saving if x87 state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
    /* might avoid saving if SSE state is not modified and MODOPT = 1 */
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
    /* might avoid saving if AVX state is not modified and MODOPT = 1 */
FI;

XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;1
XCOMP_BV field in XSAVE header ← COMPMASK;

```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVES:    void _xsaves( void *, unsigned __int64);
XSAVES64:  void _xsaves64( void *, unsigned __int64);

```

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check

1. If MXCSR does not have its initial value of 1F80H, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1, regardless of the value of XINUSE[1].

exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0. If CR4.OSXSAVE[bit 18] = 0. If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSETBV—Set Extended Control Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 D1	XSETBV	NP	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, “Enabling the XSAVE Feature Set and XSAVE-Enabled Features,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

XCR[ECX] ← EDX:EAX;

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSETBV: void _xsetbv(unsigned int, unsigned __int64);

Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an invalid XCR is specified in ECX.</p> <p>If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.</p> <p>If an attempt is made to clear bit 0 of XCR0.</p> <p>If an attempt is made to set XCR0[2:1] to 10b.</p>
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p> <p>If 66H, F3H or F2H prefix is used.</p>

Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The XSETBV instruction is not recognized in virtual-8086 mode.
--------	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
OF 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST: `int _xtest(void);`

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.
If LOCK or 66H or F2H or F3H prefix is used.