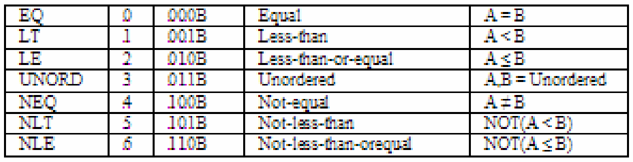
P. Specht´s „Liste der 8-Byte-Floatingpoint Befehle des masm32 Assemblers“

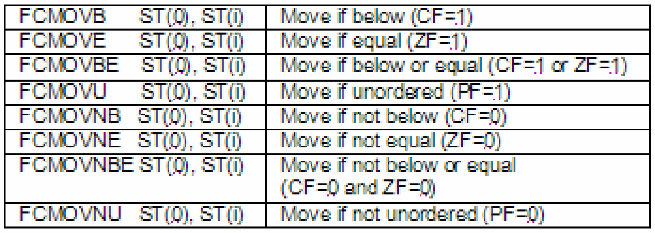
COMPACTED INTEL PENTIUM-4 PRESCOTT (*April 2004*) DPFP COMMAND SET

ADDPD	**Add Packed Double-precision Floating-Point Values (8 byte)**
ADDSD	**Add low Scalar Double-precision Floating-Point Values**
ADDSUBPD	**: Packed Double-FP Add/Subtract** in the high quadword of source stores the result in the high quadword of the destination
ANDPD	**Bitwise Logical AND of Packed Double-precision Floating-Point Values**
ANDNPD	**Bitwise Logical AND NOT of Packed Double-precision Floating-Point Values**
CMPPD	**Compare Packed Double-precision Floating-Point Values**



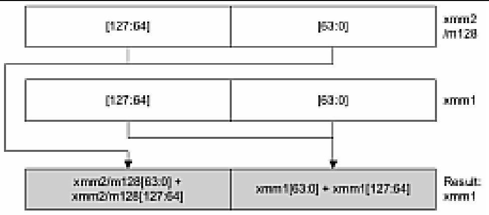
EQ	0	000B	Equal	A = B
LT	1	001B	Less-than	A < B
LE	2	010B	Less-than-or-equal	A ≤ B
UNORD	3	011B	Unordered	A,B = Unordered
NEQ	4	100B	Not-equal	A ≠ B
NLT	5	101B	Not-less-than	NOT(A < B)
NLE	6	110B	Not-less-than-orequal	NOT(A ≤ B)

CMPSD	**Compare Scalar Double-precision Floating-Point Values** = compare dword at address DS:(E)SI with dword at address ES:(E)DI;
COMISD	**Compare Scalar Ordered Double-precision Floating-Point Values and Set EFLAGS** Compare low doubleprecision floating-point values in *xmm1* and *xmm2/mem64* , set the EFLAGS
CVTDQ2PD	**Convert Packed Doubleword Integers to Packed Double-precision Floating-Point** from *xmm2/m128* to two packed double-precision floating-point values in *xmm1*.
CVTPD2DQ	**Convert Packed Double-precision Floating-Point Values to Packed Doublewd Integers** *xmm2/m128* to two packed signed doubleword integers in *xmm1*
CVTPD2PI	**Convert Packed Double-precision Floating-Point Values to Packed Doublewd Integers**
CVTPD2PS	**Convert Packed Double-precision Floating-Point Values to Packed Single-PrecisionFP**
CVTPI2PD	**Convert Packed Doubleword Integers to Packed Double-precision Floating-Point**
CVTPS2PD	**Convert Packed Single-Precision Floating-Point Values to Packed Double-precisionFP**
CVTSD2SI	**Convert Scalar Double-precision Floating-Point Value to Doubleword Integer**
CVTSD2SS	**Convert Scalar Double-precision Floating-Point Value to Scalar Single-PrecisionFP**
CVTSI2SD	**Convert Doubleword Integer to Scalar Double-precision Floating-Point Value**
CVTSS2SD	**Convert Scalar Single-Precision Floating-Point Value to Scalar Double-precision FP**
CVTTPD2PI	**Convert with Truncation Packed Double-precision FP to Packed Doubleword Integers**
CVTTPD2DQ	**Convert with Truncation Packed Double-precision FP to Packed Doubleword Integers**
CVTTSD2SI	**Convert with Truncation Scalar Double-precision FP to Signed Doubleword Integer**
DIVPD	**Divide Packed Double-precision Floating-Point Values** in *xmm1* by packed doubleprecision floating-point values *xmm2/m128*.
DIVSD	**Divide Scalar Double-precision Floating-Point Values** Divide low double-precision floating-point value in *xmm1* by low double-precision floating-point value in *xmm2/mem64*.
F2XM1	**Compute 2x–1**
FABS	**Absolute Value**
FADD	FADD *m32fp* **FADD *m64fp*** FADD ST(0), ST(i) FADD ST(i), ST(0)
FADDP	FADDP ST(i), ST(0) FADDP Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
FIADD	FIADD *m32int* FIADD *m16int*
FCHS	**Change Sign**
FCMOVcc	**Floating-Point Conditional Move**



FCMOVB	ST(0), ST(i)	Move if below (CF=1)
FCMOVE	ST(0), ST(i)	Move if equal (ZF=1)
FCMOVBE	ST(0), ST(i)	Move if below or equal (CF=1 or ZF=1)
FCMOVU	ST(0), ST(i)	Move if unordered (PF=1)
FCMOVNB	ST(0), ST(i)	Move if not below (CF=0)
FCMOVNE	ST(0), ST(i)	Move if not equal (ZF=0)
FCMOVNBE	ST(0), ST(i)	Move if not below or equal (CF=0 and ZF=0)
FCMOVNU	ST(0), ST(i)	Move if not unordered (PF=0)

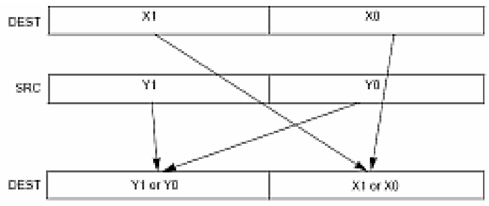
FCOS	**Cosine**
FDIV **FDIVP** **FIDIV**	**Divide**
FDIVR **FDIVRP** **FIDIVR**	**Reverse Divide**
FFREE	**Free Floating-Point Register**
FINIT **FNINIT**	**Initialize FPU after checking for pending unmasked floating-point exceptions.** **Initialize FPU without checking for pending unmasked floating-point exceptions.**
FLD	**Load Floating Point Value** Push *m32fp* onto the FPU register stack. **Push *m64fp* onto the FPU register stack.** **Push *m80fp* onto the FPU register stack.** **Push ST(i) onto the FPU register stack.**
FLD1 **FLDL2T** **FLDL2E** **FLDPI** **FLDLG2** **FLDLN2** **FLDZ**	**Load Constant**
FMUL **FMULP** **FIMUL**	**Multiply**
FNOP	**No Operation**
FPATAN	**Partial Arctangent**
FPREM	**Partial Remainder**
FPREM1	**Partial Remainder**
FPTAN	**Partial Tangent**
FRNDINT	**Round to Integer**
FSCALE	**Scale**
FSIN	**Sine**
FSINCOS	**Sine and Cosine**
FSQRT	**Square Root**
FST **FSTP**	**Store Floating Point Value** Copy ST(0) to *m32fp*. **Copy ST(0) to *m64fp*.** **Copy ST(0) to ST(i).** Copy ST(0) to *m32fp* and pop register stack. **Copy ST(0) to *m64fp* and pop register stack.** **Copy ST(0) to *m80fp* and pop register stack.** **Copy ST(0) to ST(i) and pop register stack.**
FSUB **FSUBP** **FISUB**	**Subtract**
FSUBR **FSUBRP** **FISUBR**	**Reverse Subtract**
FTST	**TEST**
 FUCOM **FUCOMP** **FUCOMPP**	**Unordered Compare Floating Point Values** Compare ST(0) with ST(i). Compare ST(0) with ST(1). Compare ST(0) with ST(i) and pop register stack. Compare ST(0) with ST(1) and pop register stack. Compare ST(0) with ST(1) and pop register stack twice.
FXAM	**ExamineModR/M**
FXCH	**Exchange Register Contents**
FXRSTOR	Restore x87 FPU, MMX Technology, SSE, SSE2, and SSE3 State
FXSAVE	Save x87 FPU, MMX Technology, SSE, and SSE2 State
FXTRACT	**Extract Exponent and Significand**
FYL2X	**Compute y * log2x**
FYL2XP1	**Compute y * log2(x +1)**
HADDPD	**Packed Double-FP Horizontal Add**

	 [127:64] [63:0] xmm2 /m128 [127:64] [63:0] xmm1 xmm2/m128[63:0] + xmm2/m128[127:64] xmm1[63:0] + xmm1[127:64] Result: xmm1
HSUBPD	**: Packed Double-FP Horizontal Subtract** **like HADDPD, but subtract from lower the upper**
Jcc	**Jump if Condition Is Met *)**
JMP	**Jump**
LDDQU	**: Load Unaligned Integer 128 Bits**
LEA	**Load Effective Address**
LOOP **LOOPcc**	**Loop According to ECX Counter** E2 *cb* LOOP *rel8* Decrement count; jump short if count ≠ 0. E1 *cb* LOOPE *rel8* Decrement count; jump short if count ≠ 0 and ZF = 1. E0 *cb* LOOPNE *rel8* Decrement count; jump short if count ≠ 0 and ZF = 0.
MASKMOVDQU	**Store Selected Bytes of Double Quadword** Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:EDI.
MASKMOVQ	**Store Selected Bytes of Quadword**
MAXPD	**Return Maximum Packed Double-precision Floating-Point Values** Return the maximum double-precision floating-point values between *xmm2/m128* and *xmm1*. 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.
MAXSD	**Return Maximum Scalar Double-precision Floating-Point Value** Return the maximum scalar doubleprecision floating-point value between *xmm2/mem64* and *xmm1*
MINPD	**Return Minimum Packed Double-precision Floating-Point Values (see MAXPD)**
MINSD	**Return Minimum Scalar Double-precision Floating-Point Value (see MAXSD)**
MOV	

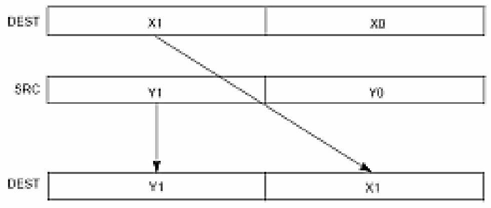
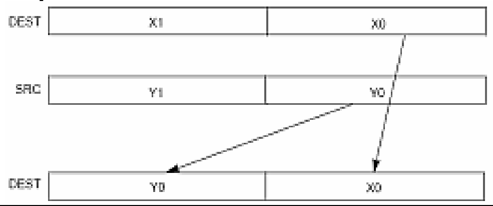
MOV	*r/m8,r8*	Move	*r8*	to	*r/m8.*
MOV	*r/m16,r16*	Move	*r16*	to	*r/m16.*
MOV	*r/m32,r32*	Move	*r32*	to	*r/m32.*
MOV	*r8,r/m8*	Move	*r/m8*	to	*r8.*
MOV	*r16,r/m16*	Move	*r/m16*	to	*r16.*
MOV	*r32,r/m32*	Move	*r/m32*	to	*r32.*
MOV	*r/m16,Sreg***	Move	segment	register to	*r/m16.*
MOV	*Sreg,r/m16***	Move	*r/m16*	to segmt	register
MOV	AL,*moffs8**	Move	byte	at (*seg:offset*)	to AL
MOV	AX,*moffs16**	Move	word	at(*seg:offset*)	to AX
MOV	EAX,*moffs32**	Move	doubleword	at(*seg:offset*)	to EAX
MOV	*moffs8*,AL	Move	AL	to(*seg:offset*).	
MOV	*moffs16**,AX	Move	AX	to(*seg:offset*).	
MOV	*moffs32**,EAX	Move	EAX	to(*seg:offset*).	
MOV	*r8, imm8*	Move	*imm8*	to	*r8.*
MOV	*r16, imm16*	Move	*imm16*	to	*r16.*
MOV	*r32, imm32*	Move	*imm32*	to	*r32.*
MOV	*r/m8, imm8*	Move	*imm8*	to	*r/m8.*
MOV	*r/m16, imm16*	Move	*imm16*	to	*r/m16.*
MOV	*r/m32, imm32*	Move	*imm32*	to	*r/m32.*

MOVAPD	**Move Aligned Packed Double-precision Floating-Point Values:** Move packed double-precision floating-point values from *xmm2/m128* to *xmm1*. Move packed double-precision floating-point values from *xmm1* to *xmm2/m128*
MOVD **MOVQ**	Move Doubleword **Move Quadword**
MOVDDUP	**: Move One Double-FP and Duplicate** Move (lower) double-precision floatingpoint value from the lower 64-bit operand in *xmm2/m64* to *xmm1* and duplicate to the upper 64 bit of xmm1
MOVDQA	**Move Aligned Double Quadword**
MOVDQU	**Move Unaligned Double Quadword**
MOVDQ2Q	**Move low Quadword from XMM to MMX Technology Register**
MOVHPD	**Move High64bit of Packed-Double-precision Floating-Point Value**

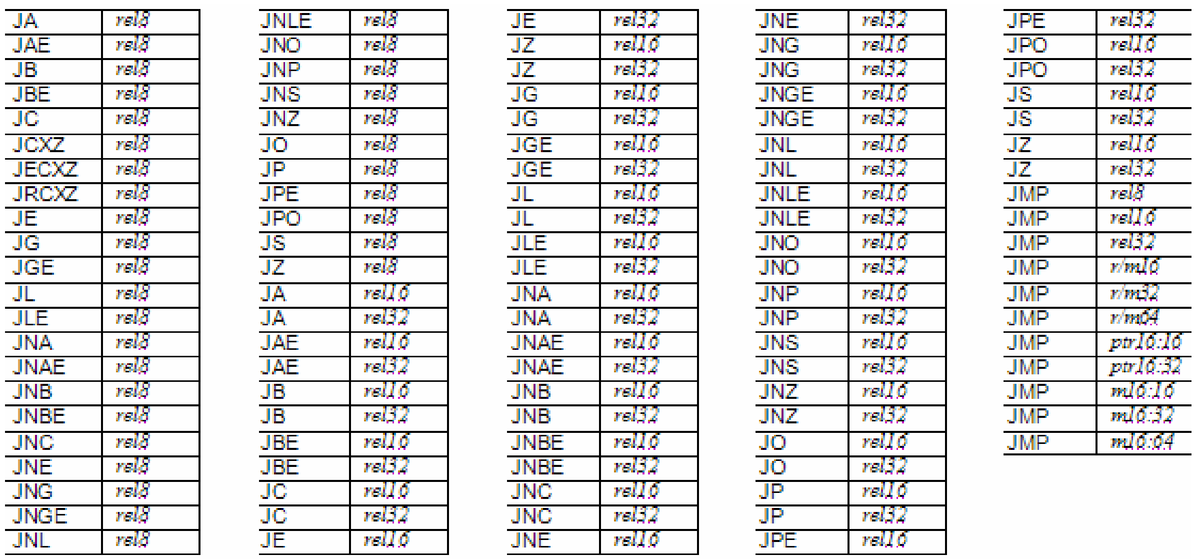
	Move double-precision floating-point value from *m64* to high quadword of *xmm*. Move double-precision floating-point value from high quadword of *xmm* to *m64*.
MOVLPD	**Move Low Packed Double-precision Floating-Point Value (see MOVHPD)**
MOVMSKPD	**Extract Packed Double-precision Floating-Point Sign Mask** Extract 2-bit sign mask from *xmm* and store in *r32*.
MOVNTDQ	**Store Double Quadword Using Non-Temporal Hint** Move double quadword from *xmm* to *m128* using non-temporal hint.
MOVNTPD	**Store Packed Double-precision Floating-Point Values Using Non-Temporal Hint**
MOVQ	**Move Quadword**
MOVQ2DQ	**Move Quadword from MMX Technology to low quadword of XMM Register**
MOVSD	**Move Scalar Double-precision Floating-Point Value** Move scalar double-precision floating-point value from *xmm2/m64* to *xmm1* register. Move scalar double-precision floating-point value from *xmm1* register to *xmm2/m64*.
MOVUPD	**Move Unaligned Packed Double-precision Floating-Point Values** Move packed double-precision floating-point values from *xmm2/m128* to xmm1. Move packed double-precision floating-point values from xmm1 to *xmm2/m128*.
MOVZX	**Move with Zero-Extend**
MULPD	**Multiply Packed Double-precision Floating-Point Values** in *xmm2/m128* by *xmm1*
MULSD	**Multiply Scalar Double-precision Floating-Point Values** Multiply the low double-precision floating-point value in *xmm2/mem64* by low double-precision floating-point value in *xmm1*.
ORPD	**Bitwise Logical OR of Double-precision Floating-Point Values** in *xmm2/m128* and *xmm1*.
PAND	**Logical AND** Bitwise AND *mm/m64* and *mm*. Bitwise AND of *xmm2/m128* and *xmm1*. The destination operand can be an MMX technology register or an XMM register.
PANDN	**Logical AND NOT** (see above)
PAUSE	**Spin Loop Hint**
PAVGB **PAVGW**	**Average Packed Integers**
PCMPEQB PCMPEQW **PCMPEQD**	**Compare Packed Data for Equal** Compare packed doublewords in *mm/m64* and *mmx* for equality Compare packed doublewords in *xmm2/m128* and xmm1 for equality
PMOVMSKB	**Move Byte Mask** of *xmm or mmx* to *r32*.
POP	**Pop a Value from the Stack**
PSLLDQ	**Shift** *xmm1* **Double Quadword Left Logical** by *imm8* bytes and by shifting in 0s.
PSRLDQ	**Shift Double Quadword Right Logical**
PUSH	**Push Word or Doubleword Onto the Stack**
PUSHF **PUSHFD**	**Push EFLAGS Register onto the Stack**
PXOR	**Logical Exclusive OR**
REP REPE REPZ REPNE REPNZ	**Repeat String Operation Prefix** F3 6C REP INS *m8*, DX Valid Valid Input (E)CX bytes from port DX into ES:[(E)DI]. F3 6C REP INS *m8*, DX Valid N.E. Input RCX bytes from port DX into [RDI]. F3 6D REP INS *m16*, DX Valid Valid Input (E)CX words from port DX into ES:[(E)DI.] **F3 6D REP INS *m32*, DX Valid Valid Input (E)CX doublewords from port DX into ES:[(E)DI].** F3 A4 REP MOVS *m8, m8* Valid Valid Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]. F3 A5 REP MOVS *m16, m16* Valid Valid Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]. **F3 A5 REP MOVS *m32, m32* Valid Valid Move (E)CX doublewords fromDS:[(E)SI] to ES:[(E)DI].** F3 6E REP OUTS DX, *r/m8* Valid Valid Output (E)CX bytes from DS:[(E)SI] to port DX. F3 6F REP OUTS DX, *r/m16* Valid Valid Output (E)CX words from DS:[(E)SI] to port DX. F3 AC REP LODS AL Valid Valid Load (E)CX bytes from DS:[(E)SI] to AL. F3 AD REP LODS AX Valid Valid Load (E)CX words from DS:[(E)SI] to AX. **F3 AD REP LODS EAX Valid Valid Load (E)CX doublewords from DS:[(E)SI] to EAX.**
RET	**Return from Procedure**
SHLD	**Double-precision Shift Left** Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right. Shift *r/m16* to left CL places while shifting bits from *r16* in from the right. Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right. Shift *r/m32* to left CL places while shifting bits from *r32* in from the right.
SHRD	**Double-precision Shift Right** Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left.

	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left. Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left. Shift *r/m32* to right CL places while shifting bits from *r32* in from the left.
SHUFPD	**Shuffle Packed Double-precision Floating-Point Values** Shuffle packed double-precision floating-point values selected by *imm8* from *xmm1* and *xmm2/m128* to *xmm1*.  DEST X1 X0 SRC Y1 Y0 DEST Y1 or Y0 X1 or X0 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.
SQRTSD	**Compute Square Root of Scalar Double-precision Floating-Point Value** Computes square root of the low double-precision floatingpoint value in *xmm2/m64* and stores the results in *xmm1*.

STOS
STOSB
STOSW
STOSD
	Store String
	store AL at address ES:(E)DI;
	store AX at address ES:(E)DI;
	store AL at address ES:(E)DI;
	store AX at address ES:(E)DI;
	store EAX at address ES:(E)DI;

SUBPD	**Subtract Packed Double-precision Floating-Point Values** in *xmm2/m128* from *xmm1*
SUBSD	**Subtract Scalar Double-precision Floating-Point Values** in *xmm2L/m64* from *xmm1* Subtracts the low double-precision floating-point values in *xmm2/mem64* from *xmm1*.
UCOMISD	**Unordered Compare Scalar Double-precision Floating-Point Values** n *xmm1* and *xmm2/m64* **and Set EFLAGS** Performs and 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). Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 64 bit memory location.
UNPCKHPD	**Unpack and Interleave High Packed Double-precision Floating-Point Values** UNPCKHPD *xmm1*, *xmm2/m128* 
UNPCKLPD	**Unpack and Interleave Low Packed Double-precision Floating-Point Values** 
VERR VERW	Verify a Segment for Reading Verify a Segment for Writing 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.
WAIT = **FWAIT**	Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding
XORPD	**Bitwise Logical XOR for Double-precision Floating-Point Values** 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.



JA	*rel8*	JNLE	*rel8*	JE	*rel32*	JNE	*rel32*	JPE	*rel32*
JAE	*rel8*	JNO	*rel8*	JZ	*rel16*	JNG	*rel16*	JPO	*rel16*
JB	*rel8*	JNP	*rel8*	JZ	*rel32*	JNG	*rel32*	JPO	*rel32*
JBE	*rel8*	JNS	*rel8*	JG	*rel16*	JNGE	*rel16*	JS	*rel16*
JC	*rel8*	JNZ	*rel8*	JG	*rel32*	JNGE	*rel32*	JS	*rel32*
JCXZ	*rel8*	JO	*rel8*	JGE	*rel16*	JNL	*rel16*	JZ	*rel16*
JECXZ	*rel8*	JP	*rel8*	JGE	*rel32*	JNL	*rel32*	JZ	*rel32*
JRCXZ	*rel8*	JPE	*rel8*	JL	*rel16*	JNLE	*rel16*	JMP	*rel8*
JE	*rel8*	JPO	*rel8*	JL	*rel32*	JNLE	*rel32*	JMP	*rel16*
JG	*rel8*	JS	*rel8*	JLE	*rel16*	JNO	*rel16*	JMP	*rel32*
JGE	*rel8*	JZ	*rel8*	JLE	*rel32*	JNO	*rel32*	JMP	*r/m16*
JL	*rel8*	JA	*rel16*	JNA	*rel16*	JNP	*rel16*	JMP	*r/m32*
JLE	*rel8*	JA	*rel32*	JNA	*rel32*	JNP	*rel32*	JMP	*r/m64*
JNA	*rel8*	JAE	*rel16*	JNAE	*rel16*	JNS	*rel16*	JMP	*ptr16:16*
JNAE	*rel8*	JAE	*rel32*	JNAE	*rel32*	JNS	*rel32*	JMP	*ptr16:32*
JNB	*rel8*	JB	*rel16*	JNB	*rel16*	JNZ	*rel16*	JMP	*m16:16*
JNBE	*rel8*	JB	*rel32*	JNB	*rel32*	JNZ	*rel32*	JMP	*m16:32*
JNC	*rel8*	JBE	*rel16*	JNBE	*rel16*	JO	*rel16*	JMP	*m16:64*
JNE	*rel8*	JBE	*rel32*	JNBE	*rel32*	JO	*rel32*		
JNG	*rel8*	JC	*rel16*	JNC	*rel16*	JP	*rel16*		
JNGE	*rel8*	JC	*rel32*	JNC	*rel32*	JP	*rel32*		
JNL	*rel8*	JE	*rel16*	JNE	*rel16*	JPE	*rel16*		