# x86 Basic Instructions, no x87FPU, MMX, SSE

## Fixed Point Arithmetic Operations (Addition, Subtraction, Comparison, Multiplication and Division)

| | | |
|---|---|---|
| **ADD** | **O S Z A C P** | **Add without Carry**, Adds the destination and source operand and stores the result in the destination operand. The operands (dest \ source) can be Reg 8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32. If Imm is used as operand, and the Imm value is shorter than the destination operand, the Imm value is sign extended. |
| **ADC** | **O S Z A C P** | **Add with Carry**, Adds CF, source and destitution operand and stores the result in the destination operand. The operands (dest \ source) can be Reg8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32.If Imm is used as operand, and the Imm value is shorter than the destination operand, the Imm value is sign extended. ADC EAX, 0 increases EAX by 1 if CF is set. |
| **SUB** | **O S Z A C P** | **Subtract**, Subtracts the source from the destination operand and stores the result in the destination operand. The operands (dest \ source) can be Reg 8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32. If Imm is used as operand, and the Imm value is shorter than the destination operand, the Imm value is sign extended. |
| **CMP** | **O S Z A C P** | **Compare Two Operands**, Compares the first operand with the second operand by subtracting the second operand from the first operand without storing the result. The operands (dest \ source) can be Reg8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32.If Imm is used as operand, and the Imm value is shorter than the destination operand, the Imm value is sign extended. All flags are set according to the result of the subtraction which isn't stored. |
| **SBB** | **O S Z A C P** | **Subtract with Carry**, Subtracts the source and the CF from the destination operand and stores the result in the destination operand. The operands (dest \ source) can be Reg 8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32. If Imm is used as operand, and the Imm value is shorter than the destination operand, the Imm value is sign extended. SBB EAX, 0 decreases EAX by 1 if CF is set |
| **MUL** | **O** s z a **C** p | **Unsigned Multiply**, Performs an unsigned multiplication of AL\AX\EAX by Reg\Mem 8\16\32 in the operand and stores the result in the AH:AL\DX:AX\EDX:EAX register pair. If the upper half of the result is 0, OF and CF are set to 0, if the upper half of the result≠0, OF and CF are set to 1 |
| **IMUL** | **O** s z a **C** p | **Signed Multiply**, Performs an signed multiplication of one, two or three operands. |
| | | **One Operand:** Performs a signed multiplication of AL\AX\EAX by Reg\Mem 8\16\32 in the operand and stores the result in the AH:AL\DX:AX\EDX:EAX register pair. If the upper half of the result is 0, OF and CF are set to 0, if the upper half of the result≠0, OF and CF are set to 1 |
| | | **Two Operands:** The first operand Reg 16\32 is signed multiplied by the second operand Reg\Mem 16\32 or Imm8 and the result is stored in the first operand. If the result is greater than the first operand, only the lower part of the result is stored in the destination operand and CF and OF are set to 1. If the result fits in the destination operand, OF and CF are 0. |
| | | **Three Operands:** The first operand Reg 16\32 is the result of the signed multiplication of the second operand Reg\Mem 16\32 and the third operand Imm 8\16\32. If the result is greater than the first operand, only the lower part of the result is stored in the destination operand and CF and OF are set to 1. If the result fits in the destination operand, OF and CF are 0. |
| **DIV** | o s z a c p | **Unsigned Divide modulo**, Performs an division of AH:AL\DX:AX\EDX:EAX modulo the Reg\Mem 8\16\32 in the operand and stores the quotient in AL\AX\EAX register and the remainder in the AH\DX\EDX register. The operation fails with a divide error if the quotient overflows the AL\AX\EAX register. All flags remain unchanged. If AH:AL = 0x01FD and BL = 0xFF, DIV BL yields AH = 0x1FD mod 0xFF = 0xFE and AL = 0x1FD ÷ 0xFF = 1. |
| **IDIV** | o s z a c p | **Signed Divide**, Performs an signed division of AH:AL\DX:AX\EDX:EAX modulo the Reg\Mem 8\16\32 in the operand and stores the quotient in AL\AX\EAX register and the remainder in the AH\DX\EDX register. The remainder always keeps the sign of the dividend. The operation fails with a divide error if the quotient overflows the AL\AX\EAX register. All flags remain unchanged. If AH:AL = 0xFFF2 / -14 and BL = 0x04, DIV BL yields AH = 0xFFF2 / -14 mod 0x04 = 0xFE / -2 and AL = 0xFFF2 / -14 ÷ 0x04 = 0xFD / -3. |
| **INC** | **O S Z A** c **P** | **Increment by 1**, Increases the operand Reg\Mem 8\16\32 by one. The CF is not changed, while the other flags are set depending on the result of the operation. If a Reg8\16\32 should be incremented depending on the C-Flag, ADC Reg8\16\32, 0 is faster. |
| **DEC** | **O S Z A** c **P** | **Decrement by 1**, Decreases the operand Reg\Mem 8\16\32 by one. The CF is not changed, while the other flags are set depending on the result of the operation. If a Reg8\16\32 should be decremented depending on the C-Flag, SBB Reg8\16\32, 0 is faster. |
| **XADD** | **O S Z A C P** | **Exchange and Add**, Adds the first and second operand and saves the result in a temporary register. Then the Reg\Mem 8\16\32 from the destination register is written to the source operand Reg 8\16\32. Finally the result of the addition in the temporary register is saved in the destination register. All flags are set according to the result in the destination operand. |

## Logic operations like AND, OR, XOR, NOT and NEG as well as Bit shifting, rolling, counting and scanning

| | | |
|---|---|---|
| **AND** | **OF** and **CF** = 0<br>**SF** = MSB of dest op.<br>**PF** is set if 2n set bits in destination,<br>**AF** unaffected | **Logical AND**, Performs a bitwise AND between destination and source operand and stores the result in the destination operand. The operands (dest \ source) can be Reg8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Mem 8\16\32. |
| **OR** | | **Logical OR**, Performs a bitwise OR between the first and second operand and stores the result in the destination operand. The operands (dest \ source) can be Reg 8\16\32 and Reg\Mem\Imm 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32 |
| **XOR** | | **Logical XOR**, Performs a bitwise XOR between the first and second operand and stores the result in the destination operand. The operands (dest \ source) can be Reg 8\16\32 and Reg\Mem\Imm 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32 |
| **TEST** | | **Logical Compare**, Performs a bitwise AND between destination and source operand without storing the result in the destionation operand. The operands (dest \ source) can be Reg8\16\32 and Imm\Reg\Mem 8\16\32 or Mem 8\16\32 and Reg\Mem 8\16\32. The SF, ZF and PF are set according to the result. |
| **NEG** | **O S Z A C P** | **Two's Complement Negation**, Performs a two's complement on the Reg\Mem 8\16\32 in the operand. The operation inverts all bits in the operand by XORing them by 1 and then adds 1 to the result of the XOR operation. The operation changes the sign of signed binary values in which +127 is represented by 0x7F, +1 by 0x01, 0 by 0x00, -1 by 0xFF, -2 by FE and -127 by 0x80. The one's complement NOT is used to change the sign in binary values which 0 is represented both by 0x00 and 0xFF. So the two's complement of -19 or 0xED is +19 or 0x13. |
| **NOT** | o s z a c p | **One's Complement Negation**, Performs a one's complement on the Reg\Mem 8\16\32 in the operand. The operation inverts all bits in the operand by XORing them by 1. The operation changes the sign of signed binary values in which +127 is represented by 0x7F, +1 by 0x01, zero is represented both by 0x00 and 0xFF, -1 by 0xFE and -127 by 0x80. |
| **ROL**<br>**ROR** | **OF** is defined in 1-bit shifts by MSB XOR CF<br>**CF** is the bit shifted out of | **Rotate register**, Rotates the Reg\Mem 8\16\32 in the first operand by the number of bits specified by Imm8 or the CL register in the second operand without using the carry flag. Only the last 5 bits of the second operand are used which limits the the rotation of the first operand to 0-31 bits. ROR is used for right rotations, while ROL is used for left rotations. The CF is defined by the state of the MSB/LSB after a shift, the ROL instruction takes the CF from the MSB, while the ROR instruction takes the LSB. The OF is defined in 1-bit shift operations as the result of the MSB of the destination operand XOR CF. |

| | | |
|---|---|---|
| **RCL**<br>**RCR** | the register,<br>s z a p | **Rotate register trough Carry Flag**, Rotates the Reg\Mem 8\16\32 in the first operand by the number of bits specified by Imm8 or the CL register in the second operand without using the carry flag. Only the last 5 bits of the second operand are used which limits the the rotation of the first operand to 0-31 bits. ROR is used for right rotations, while ROL is used for left rotations. The OF is defined in 1-bit shift operations as the result of the MSB of the destination operand XOR CF. |
| **SAL**<br>**SHL** | **OF** is only set in 1-bit shifts if the sign changes,<br>**CF** is the bit shifted out of the register,<br>**S Z** a **P** | **Shift Left**, The Reg\Mem 8\16\32 in the first operand is shifted to the left by the number of bits specified by Imm8 or the CL register in the second operand. The amount of shifted bits is filled with zeros from the right, while the last bit which is shifted left-out of the register is stored in the CF. The OF is defined in 1-bit shift operations as the result of the MSB of the destination operand XOR CF. The operation can be used to multiply the first operand by $2^n$, n is the value in the second operand. |
| **SAR** | | **Arithmetic Shift Right**, The Reg\Mem 8\16\32 in the first operand is shifted to the right by the number of bits specified by Imm8 or the CL register in the second operand. The MSB is kept as sign and the amount of shifted bits is inserted as zeros right next to the MSB, while the last bit which is shifted right-out of the register is stored in the CF. The OF is defined in 1-bit shift operations as the MSB of the destination operand. The operation can be used to divide the signed value in first operand by $2^n$, n is the value in the second operand. |
| **SHR** | | **Logical Shift Right**, The Reg\Mem 8\16\32 in the first operand is shifted to the right by the number of bits specified by Imm8 or the CL register in the second operand. The amount of shifted bits is inserted as zeros from the left, while the last bit which is shifted left-out of the register is stored in the CF. The OF is defined in 1-bit shift operations as the MSB of the destination operand. The operation can be used to divide unsigned values in first operand by $2^n$, n is the value in the second operand. |
| **SHLD** | **OF** is only set in 1-bit shifts, if the sign changes,<br>**CF** is the LSB shifted out of the register,<br>**S Z** a **P** | **Double Precision Shift Left**, The Reg\Mem 16\32 in the first operand and the Reg 16\32 of the second operand is shifted to the left by the number of bits specified by Imm8 or the CL register in the third operand. The amount of shifted bits is inserted as zeros from the right into the second operand and the bits are shifted trough the second into the first operand, while the last bit which is shifted left-out of the first operand is stored in the CF. |
| **SHRD** | | **Double Precision Shift Right**, The Reg\Mem 16\32 in the first operand and the Reg 16\32 of the second operand is shifted to the right by the number of bits specified by Imm8 or the CL register in the third operand. The amount of shifted bits is inserted as zeros from the left into the second operand and the bits are shifted trough the second into the first operand, while the last bit which is shifted right-out of the first operand is stored in the CF. |
| **BSF** | o s **Z** a c p | **Bit Scan Forward**, Scans the source operand for the **least significant set bit**, if found, the bit-index (number of the set bit) is stored in the destination operand. The operands (dest \ source) can be Reg16\32 and Reg\Mem 16\32. If the source operand is zero, the ZF is set and the dest operand is undefined. |
| **BSR** | o s **Z** a c p | **Bit Scan Reverse**, Scans the source operand for the **most significant set bit**, if found, the bit-index (number of the set bit) is stored in the destination operand. The operands (dest \ source) can be Reg16\32 and Reg\Mem 16\32. If the source operand is zero, the ZF is set and the dest operand is undefined. |
| **POPCNT** | **O=0 S=0 Z A=0 C=0 P=0** | **Count the Number of Set Bits / Horizontal Checksum**, Counts the number of set bits in a Reg\Mem 16\32 location of the second operand and stores the result in the Reg16\32 of the first operand. The OF, SF AF, CF and PF are cleared, ZF is cleared if the result is zero. |
| **BT** | o s z a **C** p<br>C-Flag is a bit from the first operand, the bit position is selected by the second operand. | **Bit Test**, Selects a bit from Reg\Mem 16\32 (first operand) and a bit position form Reg\Mem 16\32 or Imm8 (second operand) and stores the bit in the C-Flag. If bit position is greater than the size of the first operand, the captured bit position is modulo the size of the first operand. So if the first operand is a 32 bit register and the bit position is 38, bit #6 is taken, 38 mod 32 = 6) |
| **BTS** | | **Bit Test and Set**, Selects a bit from the first operand Reg\Mem 16\32 and a bit position form the second operand Reg\Mem 16\32 or Imm8 and stores the bit in the C-Flag. The selected bit in the first operand is then set to 1. If the bit position is greater than the size of the first operand, the position is modulo the size of the first operand. So if the first operand is a 16 bit register, bit position is 59, so bit #11 is taken, 59 mod 36 = 11 |
| **BTR** | | **Bit Test and Reset**, Selects a bit from a Reg\Mem 16\32 in the first operand and a bit position form a Reg\Mem 16\32 or Imm8 in the second operand and stores the bit in the C-Flag. The selected bit in the first operand is then set to 0. If thebit position is greater than the size of the first operand, the position is modulo the size of the first operand. So if the first operand is a 16 bit register, bit position is 59, so bit #11 is taken, 59 mod 36 = 11 |
| **BTC** | | **Bit Test and Complement**, Selects a bit from the first operand Reg\Mem 16\32 and a bit position form the second operand Reg\Mem 16\32 or Imm8 and stores the bit in the C-Flag. The bit selected in the first operand is then complemented. If bit position is greater than the size of the first operand, the bit position is modulo the size of the first operand. So if the first first operand is a 16 bit register, bit position is 59, so bit #11 is taken, 59 mod 36 = 11 |

## Data Conversion Instructions

| | | |
|---|---|---|
| **AAA** | o s z **A C** p | **ASCII Adjust After Addition**, Converts the sum of 2 unpacked BCD values in AX to proper unpacked BCD values written back to AX. If the low nipple in the AL register is bigger than 9 or AF is set, AF and CF are set, 6 is added to AL and AH is incremented by 1. This causes an overflow in the low nipple of AL which ends up with the correct LSD of the BSD value in the low nipple of AL. The high nipple of AL is cleared. Eg. AX=0x101F, AAA yields AH = AH+1 = 0x11, AL = (AL+5) AND 0x0F = 0x05 |
| **DAA** | o **S Z A C P** | **Decimal Adjust AL after Addition**, Converts the sum of 2 packed BCD values in AL to proper packed BCD values written back to AL. If the low nipple of AL is greater than 9 or AF is set, 6 is added to AL and AF is set, so that a the low nipple overflows and causes a carry into the higher nipple. If AL is greater than 0x99, 0x60 is added to AL, so that AL overflows and CF is set. |
| **AAS** | o s z **A C** p | **ASCII Adjust AL After Subtraction**, Converts the subtraction result of 2 unpacked BCD values in AX to proper unpacked BCD values written to AX. If the low nipple in the AL register is bigger than 9 or AF is set, AF and CF are set, 6 is subtracted from AL and AH is decremented by 1. This causes an overflow in the low nipple of AL which ends up with the correct LSD of the BSD value in the low nipple of AL. The high nipple of AL is cleared. |
| **DAS** | o **S Z A C P** | **Decimal Adjust AL after Subtraction**, Converts the subtraction result of 2 packed BCD values in AL to proper packed BCD values written back to AL. If the low nipple of AL is greater than 9 or AF is set, 6 is subtracted from AL and AF is set, so that a the low nipple overflows and causes a carry into the higher nipple. If AL is greater than 0x99, 0x60 is subtracted from AL, so that AL overflows and CF is set. |
| **AAM** | o **S Z** a c **P** | **Convert Binary to BCD**, Converts a binary value in AL to two unpacked BCD digits in AH (MSD) and AL (LSD) with a base specified by the imm8 byte. SF, ZF and PF are set according to the result in AL. (imm8=0x0A is for decimal and imm8=0x08 for octal base) Eg. AAM 8 sets AH=AL÷ 8 and AL=AL mod 8 |
| **AAD** | o **S Z** a c **P** | **Convert BCD to Binary**, Converts 2 unpacked BCD digits in AH (MSD) and AL with a base specified by the imm8 byte to a binary value in AL, AH is set to zero. SF, ZF and PF are set according to the result in AL. (imm8=0x0A is for decimal and imm8=0x08 for octal base) Eg. AAD 8 sets AL to AL+(8*AH) and AH=0 |
| **BSWAP** | o s z a c p | **Byte Swap**, Swaps bits 7-0 with bits 31-24 and bits 15-8 with bits 23-16, This instruction converts little endian values to big endian values. |
| **CBW**<br>**CWDE** | o s z a c p | **Convert Byte to Word / Convert Word to Dword**, Sign extends the byte in AL to a word in AX or the word in AX to a doubleword in EAX. The sign or MSB of AL\AX is copied to every bit in AH or the high word of EAX. |
| | | |

| | | |
|---|---|---|
| **CWD**<br>**CDQ** | | Convert Word to Doubleword<br>Convert Doubleword to Quadword |

## String instructions

| | | |
|---|---|---|
| **MOVS**<br>**MOVSB**<br>**MOVSW**<br>**MOVSD** | o s z a c p | **Move Data from String to String**, Copies the byte (MOVSB), word (MOVSW) or dword (MOVSD) from the source memory location pointed by DS:ESI in 32 bit code or by DS:SI in 16 bit code to the destination memory location pointed by ES:EDI or ES:DI. The assumed ES segment prefix of the source operand can be overridden by specifying a segment like CS, DS, SS, FS or GS. After the operation, the (E)SI and (E)DI registers are incremented (if DF=1) or decremented (if DF=0) by the size of the copied value. Like in the other string operations INS, OUTS, LODS and STOS, the REP prefix can be used to repeat the MOVS instruction until the ECX counter is decremented to zero. All flags remain unchanged. So if ECX=20, REP MOVS byte ptr [EDI], byte ptr CS:[ESI] copies 20 bytes from CS:[ESI] to ES:[EDI]. |
| **LODS**<br>**LODSB**<br>**LODSW**<br>**LODSD** | | **Load String**, Copies the byte (MOVSB), word (MOVSW) or dword (MOVSD) from the source memory location pointed by DS:ESI in 32 bit code or by DS:SI in 16 bit code to the destination memory location pointed by ES:EDI or ES:DI. The assumed ES segment prefix of the source operand can be overridden by specifying a segment like CS, DS, SS, FS or GS. After the operation, the (E)SI and (E)DI registers are incremented (if DF=1) or decremented (if DF=0) by the size of the copied value. Like in the other string operations INS, OUTS, LODS and STOS, the REP prefix can be used to repeat the MOVS instruction until the ECX counter is decremented to zero. All flags remain unchanged. |
| **STOS**<br>**STOSB**<br>**STOSW**<br>**STOSD** | | **Store String**, Copies the byte (MOVSB), word (MOVSW) or dword (MOVSD) from the source memory location pointed by DS:ESI in 32 bit code or by DS:SI in 16 bit code to the destination memory location pointed by ES:EDI or ES:DI. The assumed ES segment prefix of the source operand can be overridden by specifying a segment like CS, DS, SS, FS or GS. After the operation, the (E)SI and (E)DI registers are incremented (if DF=1) or decremented (if DF=0) by the size of the copied value. Like in the other string operations INS, OUTS, LODS and STOS, the REP prefix can be used to repeat the MOVS instruction until the ECX counter is decremented to zero. All flags remain unchanged. |
| OUTS<br>OUTSB<br>OUTSW<br>OUTSD | | **Output String to Port**, |
| INS<br>INSB<br>INSW<br>INSD | | **Input from Port to String**, |
| **REP** | o s z a c p | **Repeat String Operation Prefix**, Repeat the string operations INS, OUTS, MOVS, LODS and STOS after the REP prefix until the preloaded ECX counter has reached zero. The pointer register (E)SI and (E)DI are incremented (if DF=1) or decremented (if DF=0) after each cycle by the size of the copied value. All flags remain unchanged. |
| **SCAS**<br>**SCASB**<br>**SCASW**<br>**SCASD** | | **Scan String**, |
| **CMPSX** | | **Compare String Operands**, |
| **REPZ**<br>**REPNZ** | | **Repeat String Operation Prefix**, |

## Data Transfer and Adressing

| | | |
|---|---|---|
| **MOV** | o s z a c p | **Move**, Copies the second operand into the first operand. The operands (dest \ source) can be Reg 8\16\32 and Reg\Mem\Imm 8\16\32 or Mem 8\16\32 and Reg\Imm 8\16\32. Furthermore it can be used to copy the contents of segment registers into Reg\Mem 16 or to copy a Reg\Mem 16 into a segment register. All flags remain unchanged. |
| **CMOV** | o s z a c p | **Conditional Move**, CMOV**cc** checks the state of the status flags and performs a mov operation if the condition is true. **cc** specifies the tested condition. The operands (dest \ source) can be Reg 16\32 and Reg\Mem 16\32 |

| | | | |
|---|---|---|---|
| **CMOVB**, Move if below (CF=1) | **CMOVO**, Move if overflow (OF=1) | **CMOVS**, Move if sign (SF=1) | **CMOVLE**, Move if less or equal (ZF=1 or SF≠OF) |
| **CMOVAE**, Move if above or equal (CF=0) | **CMOVNO**, Move if not overflow (OF=0) | **CMOVNS**, Move if not sign (SF=0) | **CMOVNLE**, Move if not less or equal (ZF=0 and SF=OF) |
| **CMOVE**, Move if equal \ zero (ZF=1) | **CMOVP**, Move if parity (PF=1) | **CMOVGE**, Move if greater or equal (SF=OF) | **CMOVBE**, Move if below or equal (CF=1 or ZF=1) |
| **CMOVNE**, Move if not equal (ZF=0) | **CMOVNP**, Move if not parity (PF=0) | **CMOVL**, Move if less (SF\OF) | **CMOVA**, Move if above (CF=0 and ZF=0) |

| | | |
|---|---|---|
| **MOVSX**<br>**MOVSXD** | | **Move with Sign-Extension**, |
| **MOVZX** | | **Move with Zero-Extend**, |
| **MOVBE** | | **Move Data After Swapping Bytes**, |
| **XCHG** | | **Exchange Register\Memory with Register**, |
| **CMPXCHG** | | **CMPXCHG, Compare and Exchange**, |
| | | **CMPXCHG8B**, Compare and Exchange 8 Bytes |
| **PUSH** | | **Push Word, Doubleword or Quadword Onto the Stack**, |
| **POP** | | **Pop a Value from the Stack**, |
| **PUSHA**<br>**PUSHAD** | | **Push All General-Purpose Registers**, Pushes all ge |
| **POPA**<br>**POPAD** | | **Pop All General-Purpose Registers**, |

## Status Flag Operations

| | | |
|---|---|---|
| **CLC** | o s z a **C=0** p | **Clear Carry Flag**, Clears the carry flag, carry flag is set to zero, all other flags are unaffected. |
| **CMC** | o s z a **NEG C** p | **Complement Carry Flag**, Complements the carry flag, carry flag is XOR'd by one, all other flags are unaffected. |
| **STC** | o s z a **C=1** p | **Set Carry Flag**, Sets the carry flag, carry flag is set to one, all other flags are unaffected. |
| **CLI** | o s z a c p **I=0** | **Clear Interrupt Flag**, Clears the interrupt flag, interrupt flag is set to zero, all other flags are unaffected. |
| **STI** | o s z a c p **I=1** | **Set Interrupt Flag**, Sets the interrupt flag, interrupt flag is set to one, all other flags are unaffected. |
| **CLD** | o s z a c p **D=0** | **Clear Direction Flag**, Clears the direction direction, carry flag is set to zero, all other flags are unaffected. |
| **STD** | o s z a c p **D=1** | **Set Direction Flag**, Sets the direction flag, direction flag is set to one, all other flags are unaffected. |
| **LAHF** | o s z a c p | **Save Flags into AH**, Copies the lower byte of the EFLAGS register into the AH register. The SF is stored in bit 7 of AH, ZF in bit 6, AF in bit 4, CF in bit 2 and PF in bit 0. The reserved bits 1, 3 and 5 of the EFLAGS register are also copied into AL. The Flags in the EFLAGS register remain unchanged. |
| **SAHF** | o **S Z A C P** | **Copy AH into Flags**, Copies the AH register into the lower byte of the EFLAGS register. The SF is stored in bit 7, ZF in bit 6, AF in bit 4, CF in bit 2 and PF in bit 0. The higher part of EFLAGS and the reserved bits 1, 3 and 5 are unaffected. |
| **PUSHF PUSHFD** | o s z a c p | **Push EFLAGS onto Stack**, Saves the EEFLAGS register on the stack and decrements the stackpointer ESP by the size of the saved operand. **PUSHF** only pushes the lower word of the EFLAGS register while the **PUSHFD** instruction pushes the whole 32 bit EFLAG register on the stack. Look on the frontpage of this manual to see which bit refers to which flag. |
| **POPF POPFD** | **O S Z A C P** | **Pop Stack into EFLAGS**, Loads the EFLAGS register with a word or dword from the stack and decreases the stackpointer by the size of the loaded value. **POPF** loads a word from the stack into the lower word of the EFLAGS register while the **POPFD** instruction replaces the whole EFLAGS register by a dword from the stack. Look on the frontpage of this manual to see which bit refers to which flag. |

## Branch Instructions (Jumps, Calls, Address Calculation)

| | | |
|---|---|---|
| **CALL** | | Call Procedure, The call instruction jumps to the location specified by the operand and pushes a return address onto the stack. The jump location is either absolute or relative to the current location.<br>Near Call: |
| | | |
| **IRET IRETD** | | Interrupt Return |
| **JMP** | | Jump |
| ***Jcc*** | | Jump if Condition Is Met, |
| | | |
| | | |
| **RET** | | Return from Procedure |
| **LOOP LOOP*cc*** | | Loop According to ECX Counter |
| **LEA** | | **Load Effective Address**, |
| | | |
| | | |
| RSM | | Resume from System Management Mode |
| BOUND | | Check Array Index Against Bounds |
| ENTER | | Make Stack Frame for Procedure Parameters |
| LEAVE | | High Level Procedure Exit |
| SYSCALL | | Fast System Call |
| SYSENTER | | Fast System Call |
| SYSEXIT | | Fast Return from Fast System Call |
| SYSRET | | Return From Fast System Call |
| | | |
| | | |
| | | |
| | | |
| | | |

## System Control Instructions

| | | |
|---|---|---|
| IN | | Input from Port |
| OUT | | Output to Port |
| INT | | Call to Interrupt Procedure |
| PAUSE | | Spin Loop Hint |
| HLT | | Halt |
| NOP | | No Operation |
| WAIT FWAIT | | Wait |

| | | |
|---|---|---|
| CLFLUSH | | Flush Cache Line |
| PREFETCH*h* | | Prefetch Data Into Caches |
| INVD | | Invalidate Internal Caches |
| WBINVD | | Write Back and Invalidate Cache |
| INVLPG | | Invalidate TLB Entry |
| UD2 | | Undefined Instruction |
| **Task Management, Segmentation and Virtualisation** | | |
| LGDT | | Load Global Descriptor Table Register |
| LLDT | | Load Local Descriptor Table Register |
| LIDT | | Interrupt Descriptor Table Register |
| SGDT | | Store Global Descriptor Table Register |
| SLDT | | Store Local Descriptor Table Register |
| SIDT | | Store Interrupt Descriptor Table Register |
| | | |
| LFENCE | | Load Fence |
| SFENCE | | Store Fence |
| LMSW | | Load Machine Status Word |
| SMSW | | Store Machine Status Word |
| | | |
| | | |
| | | |
| | | |
| LOCK | | Assert LOCK# Signal Prefix |
| | | |
| | | |
| MFENCE | | Memory Fence |
| MONITOR | | Set Up Monitor Address |
| | | |
| LAR | | Load Access Rights Byte |
| LDS<br>LES<br>LFS<br>LGS<br>LSS | | Load Far Pointer |
| | | |
| LSL | | Load Segment Limit |
| | | |
| LTR | | Load Task Register |
| STR | | Store Task Register |
| | | |
| | | |
| | | |
| SWAPGS | | Swap GS Base Register |
| ARPL | | Adjust RPL Field of Segment Selector |
| | | |
| | | |
| | | |
| VERR\VERW | | Verify a Segment for Reading or Writing |
| | | |
| | | |
| **Special Registers** | | |
| CLTS | | Clear Task-Switched Flag in CR0 |
| CPUID | | CPU Identification |
| RDMSR | | Read from Model Specific Register |
| RDPMC | | Read Performance-Monitoring Counters |
| RDTSC | | Read Time-Stamp Counter |
| RDTSCP | | Read Time-Stamp Counter and Processor ID |

| | | |
|---|---|---|
| WRMSR | | Write to Model Specific Register |
| XGETBV | | Get Value of Extended Control Register |
| XLAT<br>XLATB | | Table Look-up Translation |
| XRSTOR | | Restore Processor Extended States |
| XSAVE | | Save Processor Extended States |
| XSETBV | | Set Extended Control Register |

# X87 Floating Point Unit

## Control Instructions

| | | |
|---|---|---|
| FCLEX\FNCLEX | | Clear Exceptions |
| FDECSTP | | Decrement Stack-Top Pointer |
| FFREE | | Free Floating-Point Register |
| FINCSTP | | Increment Stack-Top Pointer |
| FINIT\FNINIT | | Initialize Floating-Point Unit |
| FLDENV | | Load x87 FPU Environment |
| FNOP | | No Operation |
| FRSTOR | | Restore x87 FPU State |
| FSAVE\FNSAVE | | Store x87 FPU State |
| FSTENV\FNSTENV | | Store x87 FPU Environment |
| FSTSW\FNSTSW | | Store x87 FPU Status Word |
| FXRSTOR | | Restore x87 FPU, MMX , XMM, and MXCSR State |
| FXSAVE | | Save x87 FPU, MMX Technology, and SSE State |

## Data Transfer Instructions

| | | |
|---|---|---|
| FCMOV*cc* | | Floating-Point Conditional Move |
| FILD | | Load Integer |
| FIST\FISTP | | Store Integer |
| FISTTP | | Store Integer with Truncation |
| FLD | | Load Floating Point Value |
| FLD1\FLDL2T\FLDL2E\FLDPI\FLDLG2\FLDLN2\FLDZ | | Load Constant |
| FLDCW | | Load x87 FPU Control Word |
| FST\FSTP | | Store Floating Point Value |
| FSTCW\FNSTCW | | Store x87 FPU Control Word |

## Data Conversion Instructions

| | | |
|---|---|---|
| FABS | | Absolute Value |
| FBLD | | Load Binary Coded Decimal |
| FBSTP | | Store BCD Integer and Pop |
| FXCH | | Exchange Register Contents |
| FXTRACT | | Extract Exponent and Significand |
| | | |

## Arithmetic Instructions

| | | |
|---|---|---|
| | | |
| FADD<br>FADD<br>FIADD | | Add |
| FDIV\FDIVP\FIDIV | | Divide |
| FDIVR\FDIVRP\FIDIVR | | Reverse Divide |
| FCHS | | Change Sign |
| FMUL\FMULP\FIMUL | | Multiply |
| FPREM | | Partial Remainder |
| FPREM1 | | Partial Remainder |

| | | |
|---|---|---|
| FRNDINT | | Round to Integer |
| FSUB\FSUBP\FISUB | | Subtract |
| FSUBR\FSUBRP\FISUBR | | Reverse Subtract |

## Trigonometric Instructions

| | | |
|---|---|---|
| FCOS | | Cosine |
| FPATAN | | Partial Arctangent |
| FPTAN | | Partial Tangent |
| FSIN | | Sine |

## Logarithmic and Exponential Instructions

| | | |
|---|---|---|
| F2XM1 | | Compute $2x^{-1}$ |
| FSCALE | | Scale |
| FSINCOS | | Sine and Cosine |
| FSQRT | | Square Root |
| FYL2X | | Compute $y * \log 2x$ |
| FYL2XP1 | | Compute $y * \log 2(x+1)$ |

## Comparison instructions

| | | |
|---|---|---|
| FCOMI FCOMIP FUCOMI FUCOMIP | | Compare Floating Point Values and Set EFLAGS |
| FICOM FICOMP | | Compare Integer |
| FTST | | TEST |
| FUCOM\FUCOMP\FUCOMPP | | Unordered Compare Floating Point Values |
| FXAM | | Examine ModR\M |

# MMX Instructions

### Control Instructions

| | | |
|---|---|---|
| EMMS | | Empty MMX Technology State |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# MMX and SSE Instructions

## Misc and State Management Instructions

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

## Basic Arithmetic (Addition, Subtraction, Multiplication and Division)

| | | | |
|---|---|---|---|
| **ADDPD** | SSE2 | O,U,I,P,D | **Add Packed Double-Precision Floating-Point Values**, Adds 2 floating point qwords XMM \Mem128 [127:64] and XMM \ Mem128 [63:0] from the second operand to XMM[127:64] and XMM[63:0] in the first operand. |
| **ADDPS** | SSE1 | O,U,I,P,D | **Add Packed Single-Precision Floating-Point Values**, Adds 4 floating point dwords XMM\Mem128 [127:96], XMM \ Mem128 [95:64], XMM\Mem128 [63:32], XMM\Mem128 [31:0] from the second operand to XMM[127:96], XMM[95:64], XMM[63:32], XMM[31:0] register in the first operand. |
| **ADDSD** | SSE2 | O,U,I,P,D | **Add Scalar Double-Precision Floating-Point Value**, Adds low floating point qword from the second operand XMM[63:0] \ Mem64 to the low qword XMM[63:0] in the first operand, The high qword XMM[127:64] in the first operand remains unchanged. |
| **ADDSS** | SSE1 | O,U,I,P,D | **Add Scalar Single-Precision Floating-Point Value**, Adds low floating point dword from the second operand XMM[31:0] \ Mem32 to the low dword XMM[31:0] in the first operand, XMM[127:32] of the first operand remain unchanged. |
| HADDPD | | | Packed Double-FP Horizontal Add |
| HADDPS | | | Packed Single-FP Horizontal Add |
| PADDB<br>PADDW<br>PADDD | | | Add Packed Integers |
| PADDQ | | | Add Packed Quadword Integers |
| PADDSB<br>PADDSW | | | Add Packed Signed Integers with Signed Saturation |
| PADDUSB<br>PADDUSW | | | Add Packed Unsigned Integers with Unsigned Saturation |
| PHADDW<br>PHADDD | | | Packed Horizontal Add |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| **SUBPD** | SSE2 | | **Subtract Packed Double-Precision Floating-Point Values**, |
| **SUBPS** | SSE1 | | **Subtract Packed Single-Precision Floating-Point Values**, |
| **SUBSD** | SSE2 | | **Subtract Scalar Double-Precision Floating-Point Values**, |
| **SUBSS** | SSE1 | | **Subtract Scalar Single-Precision Floating-Point Values**, |
| HSUBPD | | | Packed Double-FP Horizontal Subtract |
| HSUBPS | | | Packed Single-FP Horizontal Subtract |
| PSUBB<br>PSUBW<br>PSUBD | | | Subtract Packed Integers |
| PSUBQ | | | Subtract Packed Quadword Integers |
| PSUBSB<br>PSUBSW | | | Subtract Packed Signed Integers with Signed Saturation |
| PSUBUSB<br>PSUBUSW | | | Subtract Packed Unsigned Integers with Unsigned Saturation |
| | | | |
| | | | |
| | | | |
| | | | |

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| **MULPD** | SSE2 | | **Multiply Packed Double-Precision Floating-Point Values**, |
| **MULPS** | SSE1 | | **Multiply Packed Single-Precision Floating-Point Values**, |
| **MULSD** | SSE2 | | **Multiply Scalar Double-Precision Floating-Point Values**, |
| **MULSS** | SSE1 | | **Multiply Scalar Single-Precision Floating-Point Values**, |
| PMULDQ | | | Multiply Packed Signed Dword Integers |
| PMULHUW | | | Multiply Packed Unsigned Integers and Store High Result |
| PMULHW | | | Multiply Packed Signed Integers and Store High Result |
| PMULLD | | | Multiply Packed Signed Dword Integers and Store Low Result |
| PMULLW | | | Multiply Packed Signed Integers and Store Low Result |
| PMULUDQ | | | Multiply Packed Unsigned Doubleword Integers |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| **DIVPD** | SSE2 | | **Divide Packed Double-Precision Floating-Point Values**, |
| **DIVPS** | SSE1 | | **Divide Packed Single-Precision Floating-Point Values**, |
| **DIVSD** | SSE2 | | **Divide Scalar Double-Precision Floating-Point Values**, |
| **DIVSS** | SSE1 | | **Divide Scalar Single-Precision Floating-Point Values**, |
| | | | |
| | | | |
| | | | |

## Mixed Arithmetic (AddSub

| | | | |
|---|---|---|---|
| ADDSUBPD | SSE3 | O,U,I,P,D | Packed Double-Precision Floating-Point Add\Subtract, Adds high floating point qword XMM\Mem128[127:64] of second operand to the high qword XMM\Mem128[127:64] of first operand and subtracts the low floating point qword XMM\Mem128[63:0] of second operand from low qword XMM[63:0] of first operand. XMMdest = XMMdest[127:64] + XMM\Mem128src[127:64], XMMdest[63:0] - XMM\Mem128src[63:0] |
| ADDSUBPS | SSE3 | O,U,I,P,D | Packed Single-Precision Floating-Point Add\Subtract, Adds floating point dwords XMM\Mem128 [127:96] and XMM\Mem128[63:32] of second operand to the dwords XMM[127:96] and XMM[63:32] of first operand and subtracts the floating point dwords XMM\Mem128[95:64] and XMM\Mem128[31:0] of second operand from dwords XMM[95:64] and XMM[31-0] of first operand. XMMdest = XMMdest[127:96] + XMM\Mem128src[127:96], XMMdest[95:64] – XMM\Mem128src[95:64], XMMdest[63:32] + XMM\Mem128src[63:32], XMMdest[31:0] - XMM\Mem128src[31:0] |
| | | | |
| PMADDUBSW | | | Multiply and Add Packed Signed and Unsigned Bytes |
| PMADDWD | | | Multiply and Add Packed Integers |
| | | | |
| | | | |

## Exponential, Logarithmic and Square Root Calculation

| | | | |
|---|---|---|---|
| RSQRTPS | | | Compute Reciprocals of Square Roots of Packed Single- Precision Floating-Point Values |
| RSQRTSS | | | Compute Reciprocal of Square Root of Scalar Single- Precision Floating-Point Value |
| SQRTPS | | | Compute Square Roots of Packed Single-Precision Floating- Point Values |
| SQRTSD | | | Compute Square Root of Scalar Double-Precision Floating- Point Value |
| SQRTSS | | | Compute Square Root of Scalar Single-Precision Floating- Point Value |
| | | | |
| | | | |
| | | | |
| DPPD | | | Dot Product of Packed Double Precision Floating-Point Values |
| DPPS | | | Dot Product of Packed Single Precision Floating-Point Values |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| MPSADBW | | | Compute Multiple Packed Sums of Absolute Difference |
| | | | |
| | | | |
| | | | |
| **Data Comparison** | | | |
| | | | |
| PHADDSW | | | Packed Horizontal Add and Saturate |
| PHMINPOSUW | | | Packed Horizontal Word Minimum |
| PHSUBW<br>PHSUBD | | | Packed Horizontal Subtract |
| PHSUBSW | | | Packed Horizontal Subtract and Saturate |
| | | | |
| PSADBW | | | Compute Sum of Absolute Differences |
| **Data Shuffling** | | | |
| PSHUFD | | | Shuffle Packed Doublewords |
| PSHUFHW | | | Shuffle Packed High Words |
| PSHUFLW | | | Shuffle Packed Low Words |
| PSHUFW | | | Shuffle Packed Words |
| SHUFPD | | | Shuffle Packed Double-Precision Floating-Point Values |
| SHUFPS | | | Shuffle Packed Single-Precision Floating-Point Values |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| **Internal  Instructions:** | | | |
| | | | |
| | | | |
| | | | |
| **Data Packing Instructions:** | | | |
| PABSB<br>PABSW<br>PABSD | | | Packed Absolute Value |
| PACKSSWB<br>PACKSSDW | | | Pack with Signed Saturation |
| PACKUSDW | | | Pack with Unsigned Saturation |
| PACKUSWB | | | Pack with Unsigned Saturation |
| PALIGNR | | | Packed Align Right |
| PAVGB<br>PAVGW | | | Average Packed Integers |
| PSIGNB<br>PSIGNW<br>PSIGND | | | Packed SIGN |
| PEXTRB<br>PEXTRD<br>PEXTRQ | | | Extract Byte\Dword\Qword |
| PEXTRW | | | Extract Word |
| PMULHRSW | | | Packed Multiply High with Round and Scale |
| PUNPCKHBW<br>PUNPCKHWD | | | Unpack High Data |

| | | | |
|---|---|---|---|
| PUNPCKHDQ<br>PUNPCKHQDQ | | | |
| PUNPCKLBW<br>PUNPCKLWD<br>PUNPCKLDQ<br>PUNPCKLQDQ | | | Unpack Low Data |
| RCPPS | | | Compute Reciprocals of Packed Single-Precision Floating- Point Values |
| UNPCKHPD | | | Unpack and Interleave High Packed Double-Precision Floating-Point Values |
| UNPCKHPS | | | Unpack and Interleave High Packed Single-Precision Floating-Point Values |
| UNPCKLPD | | | Unpack and Interleave Low Packed Double-Precision Floating-Point Values |
| UNPCKLPS | | | Unpack and Interleave Low Packed Single-Precision Floating-Point Values |
| PMOVSX | | | Packed Move with Sign Extend |
| PMOVZX | | | Packed Move with Zero Extend |
| PSHUFB | | | Packed Shuffle Bytes |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

## Control Instructions

| | | | |
|---|---|---|---|
| LDMXCSR | | | Load MXCSR Register |
| STMXCSR | | | Store MXCSR Register State |
| | | | |
| | | | |
| | | | |

## Blending Instructions

| | | | |
|---|---|---|---|
| BLENDPD | SSE4 | none | Blend Packed Double Precision Floating-Point Values, The last 2 bits in the third operand Imm8[1:0] control whether the two qwords in XMM\Mem128[127:64] and XMM\Mem128[63:0] of the second operand are written to XMM[127:64] and XMM[63:0] of the first operand \ destination. If a bit in Imm8[1:0] is set, the source qword is written to the destination, if a bit in Imm8[1:0] is zero, the qword in the destination is unchanged. Imm8[1] controls bits[127:64], Imm8[0] controls bits[63:0] |
| BLENDPS | SSE4 | none | Blend Packed Single Precision Floating-Point Values, The last 4 bits in the third operand Imm8[3:0] control whether the 4 dwords in XMM\Mem128[127:96], XMM\Mem128[95:64], XMM\Mem128[63:32] and XMM\Mem128[31:0] of the second operand are written to XMM[127:96], XMM[95:64], XMM[63:32] and XMM[31:0] of the first operand \ destination. If a bit in Imm8[3:0] is set, the source qword is written to the destination, if a bit in Imm8[3:0] is zero, the qword in the destination is unchanged. Imm8[3] controls bits[127:96], Imm8[2] => bits[95:64], Imm8[1] => bits[63:32], Imm8[0] => bits[31:0], |
| BLENDVPD | SSE4 | none | Variable Blend Packed Double Precision Floating-Point Values, The 2 bits XMM[127] and XMM[63] of third operand control whether the two qwords in XMM\Mem128[127:64] and XMM\Mem128[63:0] of the second operand are written to XMM[127:64] and XMM[63:0] of the first operand \ destination. If a bit in third operand is set, the source qword is written to the destination, if a bit in the third operand is zero, the qword in the destination is unchanged. XMM[127] controls bits[127:64], XMM[63] controls bits[63:0] |
| BLENDVPS | SSE4 | none | Variable Blend Packed Single Precision Floating-Point Values, The 4 bits XMM[127], XMM[95], XMM[63] and XMM[31] of third operand control whether the 4 dwords in XMM\Mem128[127:96], XMM\Mem128[95:64], XMM\Mem128[63:32] and XMM\Mem128[31:0] of the second operand are written to XMM[127:96], XMM[95:64], XMM[63:32] and XMM[31:0] of the first operand \ destination. If a bit in the third operand is set, the source qword is written to the destination, if a bit in the third operand is zero, the qword in the destination is unchanged. XMM[127] controls bits[127:96], XMM[95] => bits[95:64], XMM[63] => bits[63:32], XMM[31] => bits[31:0], |
| PBLENDVB | | | Variable Blend Packed Bytes |
| PBLENDW | | | Blend Packed Words |
| | | | |
| EXTRACTPS | | | Extract Packed Single Precision Floating-Point Value |
| INSERTPS | | | Insert Packed Single Precision Floating-Point Value |
| MOVHLPS | | | Move Packed Single-Precision Floating-Point Values High to Low |
| MOVHPS | | | Move High Packed Single-Precision Floating-Point Values |
| MOVLHPS | | | Move Packed Single-Precision Floating-Point Values Low to High |
| MOVLPD | | | Move Low Packed Double-Precision Floating-Point Value |
| MOVLPS | | | Move Low Packed Single-Precision Floating-Point Values |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| RCPSS | | | Compute Reciprocal of Scalar Single-Precision Floating-Point Values |
| ROUNDPD | | | Round Packed Double Precision Floating-Point Values |
| ROUNDPS | | | Round Packed Single Precision Floating-Point Values |
| ROUNDSD | | | Round Scalar Double Precision Floating-Point Values |
| ROUNDSS | | | Round Scalar Single Precision Floating-Point Values |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| **Data Transfer Instructions** | | | |
| LDDQU | | | Load Unaligned Integer 128 Bits |
| MASKMOVDQU | | | Store Selected Bytes of Double Quadword |
| MASKMOVQ | | | Store Selected Bytes of Quadword |
| MOVAPD | | | Move Aligned Packed Double-Precision Floating-Point Values |
| MOVAPS | | | Move Aligned Packed Single-Precision Floating-Point Values |
| | | | |
| MOVDDUP | | | Move One Double-FP and Duplicate |
| MOVDQA | | | Move Aligned Double Quadword |
| MOVDQU | | | Move Unaligned Double Quadword |
| MOVDQ2Q | | | Move Quadword from XMM to MMX Technology Register |
| MOVHPD | | | Move High Packed Double-Precision Floating-Point Value |
| | | | |
| MOVNTDQA | | | Load Double Quadword Non-Temporal Aligned Hint |
| MOVNTDQ | | | Store Double Quadword Using Non-Temporal Hint |
| MOVNTI | | | Store Doubleword Using Non-Temporal Hint |
| MOVNTPD | | | Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint |
| MOVNTPS | | | Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint |
| MOVNTQ | | | Store of Quadword Using Non-Temporal Hint |
| OVD\MOVQ | | | Move Doubleword\Move Quadword |
| MOVQ2DQ | | | Move Quadword from MMX Technology to XMM Register |
| MOVSHDUP | | | Move Packed Single-FP High and Duplicate |
| MOVSLDUP | | | Move Packed Single-FP Low and Duplicate |
| MOVSS | | | Move Scalar Single-Precision Floating-Point Values |
| MOVUPD | | | Move Unaligned Packed Double-Precision Floating-Point Values |
| MOVUPS | | | Move Unaligned Packed Single-Precision Floating-Point Values |
| PINSRB\PINSRD\PINSRQ | | | Insert Byte\Dword\Qword |
| PINSRW | | | Insert Word |
| | | | |
| | | | |
| | | | |
| | | | |

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| **Comparison Operations** | | | |
| CMPPD | SSE2 | | Compare Packed Double-Precision Floating-Point Values |
| CMPPS | SSE1 | | Compare Packed Single-Precision Floating-Point Values |
| COMISD | SSE2 | | Compare Scalar Ordered Double-Precision Floating-<br>Point Values and Set EFLAGS |
| COMISS | SSE1 | | Compare Scalar Ordered Single-Precision Floating-<br>Point Values and Set EFLAGS |
| CVTDQ2PD | | | Convert Packed dword Integers to Packed<br>Double-Precision Floating-Point Values |
| CVTDQ2PS | | | Convert Packed dword Integers to Packed<br>Single-Precision Floating-Point Values |
| CVTPD2DQ | | | Convert Packed Double-Precision Floating-Point<br>Values to Packed dword Integers |
| CVTPD2PI | | | Convert Packed Double-Precision Floating-Point<br>Values to Packed dword Integers |
| CVTPD2PS | | | Covert Packed Double-Precision Floating-Point<br>Values to Packed Single-Precision Floating-Point Values |
| CVTPI2PD | | | Convert Packed dword Integers to Packed<br>Double-Precision Floating-Point Values |
| CVTPI2PS | | | Convert Packed Dword Integers to Packed Single-PrecisionFP Values |
| CVTPS2DQ | | | Convert Packed Single-Precision FP Values to PackedDword Integers |
| CVTPS2PD | | | Convert Packed Single-Precision FP Values to PackedDouble-Precision FP Values |
| CVTPS2PI | | | Convert Packed Single-Precision FP Values to PackedDword Integers |
| CVTSD2SI | | | Convert Scalar Double-Precision FP Value to Integer |
| CVTSD2SS | | | Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value |
| CVTSI2SD | | | Convert Dword Integer to Scalar Double-Precision FP Value |
| CVTSI2SS | | | Convert Dword Integer to Scalar Single-Precision FP Value |
| CVTSS2SD | | | Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value |
| CVTSS2SI | | | Convert Scalar Single-Precision FP Value to Dword Integer |
| CVTTPD2DQ | | | Convert with Truncation Packed Double-Precision FPValues to Packed Dword Integers |
| CVTTPD2PI | | | Convert with Truncation Packed Double-Precision FPValues to Packed Dword Integers |
| CVTTPS2DQ | | | Convert with Truncation Packed Single-Precision FPValues to Packed Dword Integers |
| CVTTPS2PI | | | Convert with Truncation Packed Single-Precision FPValues to Packed Dword Integers |
| CVTTSD2SI | | | Convert with Truncation Scalar Double-Precision FP Valueto Signed Integer |
| CVTTSS2SI | | | Convert with Truncation Scalar Single-Precision FP Valueto Dword Integer |
| | | | |
| MAXPD | | | Return Maximum Packed Double-Precision Floating-Point<br>Values |
| MAXPS | | | Return Maximum Packed Single-Precision Floating-Point<br>Values |
| MAXSD | | | Return Maximum Scalar Double-Precision Floating-Point<br>Value |
| MAXSS | | | Return Maximum Scalar Single-Precision Floating-Point Value |
| | | | |
| MINPD | | | Return Minimum Packed Double-Precision Floating-Point Values |
| MINPS | | | Return Minimum Packed Single-Precision Floating-Point Values |
| MINSD | | | Return Minimum Scalar Double-Precision Floating-Point Value |
| MINSS | | | Return Minimum Scalar Single-Precision Floating-Point Value |
| PCMPEQB<br>PCMPEQW<br>PCMPEQD | | | Compare Packed Data for Equal |
| PCMPEQQ | | | Compare Packed Qword Data for Equal |

| | | | |
|---|---|---|---|
| PCMPESTRI | | | Packed Compare Explicit Length Strings, Return Index |
| PCMPESTRM | | | Packed Compare Explicit Length Strings, Return Mask |
| PCMPISTRI | | | Packed Compare Implicit Length Strings, Return Index |
| PCMPISTRM | | | Packed Compare Implicit Length Strings, Return Mask |
| PCMPGTB<br>PCMPGTW<br>PCMPGTD | | | Compare Packed Signed Integers for Greater Than |
| PCMPGTQ | | | Compare Packed Data for Greater Than |
| | | | |
| PMAXSB | | | Maximum of Packed Signed Byte Integers |
| PMAXSD | | | Maximum of Packed Signed Dword Integers |
| PMAXSW | | | Maximum of Packed Signed Word Integers |
| PMAXUB | | | Maximum of Packed Unsigned Byte Integers |
| PMAXUD | | | Maximum of Packed Unsigned Dword Integers |
| PMAXUW | | | Maximum of Packed Word Integers |
| PMINSB | | | Minimum of Packed Signed Byte Integers |
| PMINSD | | | Minimum of Packed Dword Integers |
| PMINSW | | | Minimum of Packed Signed Word Integers |
| PMINUB | | | Minimum of Packed Unsigned Byte Integers |
| PMINUD | | | Minimum of Packed Dword Integers |
| PMINUW | | | Minimum of Packed Word Integers |
| PTEST | | | Logical Compare |
| UCOMISD | | | Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS |
| UCOMISS | | | Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS |
| | | | |
| | | | |
| | | | |
| | | | |

## Bit and Sign Operations (AND, OR, XOR, Bit shifting, )

| | | | |
|---|---|---|---|
| ANDPS<br>ANDPD | SSE1<br>SSE2 | none | Bitwise Logical AND of XMM[127:0] and XMM[127:0] \ Mem128, Performs a bitwise AND between XMM[127:0] \ Mem128 in the second operand and the XMM[127:0] in the first operand. Both instructions do the same, but ANDPD is longer and is only supported in SSE2, so better use ANDPS |
| ANDNPS<br>ANDNPD | SSE1<br>SSE2 | none | Bitwise Logical AND NOT of XMM[127:0] and XMM[127:0] \ Mem128, Performs a bitwise AND between XMM[127:0] \ Mem128 in the second operand and the bitwise inverted XMM[127:0] in the first operand. Both instructions do the same, but ANDNPD is longer and is only supported in SSE2, so better use ANDNPS. XMMdest[127:0] = ( NOT XMMdest[127:0] ) AND XMMsrc[127:0] |
| MOVMSKPD | | | Extract Packed Double-Precision Floating-Point Sign Mask |
| MOVMSKPS | | | Extract Packed Single-Precision Floating-Point Sign Mask |
| ORPD | | | Bitwise Logical OR of Double-Precision Floating-Point Values |
| ORPS | | | Bitwise Logical OR of Single-Precision Floating-Point Values |
| PAND | | | Logical AND |
| PANDN | | | Logical AND NOT |
| PMOVMSKB | | | Move Byte Mask |
| POR | | | Bitwise Logical OR |
| PSLLDQ | | | Shift Double Quadword Left Logical |
| PSLLW<br>PSLLD<br>PSLLQ | | | Shift Packed Data Left Logical |
| PSRAW<br>PSRAD | | | Shift Packed Data Right Arithmetic |
| PSRLDQ | | | Shift Double Quadword Right Logical |
| PSRLW<br>PSRLD<br>PSRLQ | | | Shift Packed Data Right Logical |
| PXOR | | | Logical Exclusive OR |
| XORPD | | | Bitwise Logical XOR for Double-Precision Floating-Point Values |
| XORPS | | | Bitwise Logical XOR for Single-Precision Floating-Point Values |