NASM 2.05 based x86 Instruction Reference

This document has been compiled on 2025-08-28.

Contents

Section 1: License

NASM is now licensed under the 2-clause BSD license, also known as the simplified BSD license.

Appendix A: x86 Instruction Reference

This appendix provides a complete list of the machine instructions which NASM will assemble, and a short description of the function of each one.

It is not intended to be an exhaustive documentation on the fine details of the instructions' function, such as which exceptions they can trigger: for such documentation, you should go to Intel's Web site, `http://developer.intel.com/design/Pentium4/manuals/`.

Instead, this appendix is intended primarily to provide documentation on the way the instructions may be used within NASM. For example, looking up `LOOP` will tell you that NASM allows `CX` or `ECX` to be specified as an optional second argument to the `LOOP` instruction, to enforce which of the two possible counter registers should be used if the default is not the one desired.

The instructions are not quite listed in alphabetical order, since groups of instructions with similar functions are lumped together in the same entry. Most of them don't move very far from their alphabetic position because of this.

A.1 Key to Operand Specifications

The instruction descriptions in this appendix specify their operands using the following notation:

- Registers: `reg8` denotes an 8-bit general purpose register, `reg16` denotes a 16-bit general purpose register, `reg32` a 32-bit one and `reg64` a 64-bit one. `fpureg` denotes one of the eight FPU stack registers, `mmxreg` denotes one of the eight 64-bit MMX registers, and `segreg` denotes a segment register. `xmmreg` denotes one of the 8, or 16 in x64 long mode, SSE XMM registers. In addition, some registers (such as `AL`, `DX`, `ECX` or `RAX`) may be specified explicitly.

- Immediate operands: `imm` denotes a generic immediate operand. `imm8`, `imm16` and `imm32` are used when the operand is intended to be a specific size. For some of these instructions, NASM allows an explicit specifier: for example, `ADD ESP,16` could be interpreted as either `ADD r/m32,imm32` or `ADD r/m32,imm8`. If the immediate value is known during the assembling of that instruction, and fits in the range of a signed byte, then recent versions of NASM automatically optimise the instruction by choosing the latter form. It is allowed to specify `ADD ESP,BYTE 16` explicitly, which is needed to indicate that optimisation in case the immediate value is not known by NASM. (Eg, if it is a symbol reference only resolved by the linker.) Conversely, using the qualifiers `STRICT WORD` or `STRICT DWORD` forces NASM to use the longer forms. There is a special case of the allowance of an `imm64` for particular x64 versions of the MOV instruction.

- Memory references: `mem` denotes a generic memory reference; `mem8`, `mem16`, `mem32`, `mem64` and `mem80` are used when the operand needs to be a specific size. Again, a specifier is needed in some cases: `DEC [address]` is ambiguous and will be rejected by NASM. You must specify `DEC BYTE [address]`, `DEC WORD [address]` or `DEC DWORD [address]` instead.

- Restricted memory references: one form of the `MOV` instruction allows a memory address to be specified *without* allowing the normal range of register combinations and effective address processing. This is denoted by `memoffs8`, `memoffs16`, `memoffs32` or `memoffs64`.

- Register or memory choices: many instructions can accept either a register *or* a memory reference as an operand. `r/m8` is shorthand for `reg8/mem8`; similarly `r/m16` and `r/m32`. On legacy x86 modes, `r/m64` is MMX-related, and is shorthand for `mmxreg/mem64`. When utilizing the x86-64 architecture extension, `r/m64` denotes use of a 64-bit GPR as well, and is shorthand for `reg64/mem64`.

A.2 Key to Opcode Descriptions

This appendix also provides the opcodes which NASM will generate for each form of each instruction. The opcodes are listed in the following way:

- A hex number, such as `3F`, indicates a fixed byte containing that number.

- A hex number followed by `+r`, such as `C8+r`, indicates that one of the operands to the instruction is a register, and the 'register value' of that register should be added to the hex number to produce the generated byte. For example, EDX has register value 2, so the code `C8+r`, when the register operand is EDX, generates the hex byte `CA`. Register values for specific registers are given in section A.2.1.

- A hex number followed by `+cc`, such as `70+cc`, indicates that the instruction name has a condition code suffix, and the numeric representation of the condition code should be added to the hex number to produce the generated byte. For example, the code `70+cc`, when the instruction contains the `NE` condition, generates the hex byte `75`. Condition codes and their numeric representations are given in section A.2.2. In one case, `70+(cc^1)` is given, which means that the inverse condition code is encoded in the byte.

- A slash followed by a digit, such as `/2`, indicates that one of the operands to the instruction is a memory address or register (denoted `mem` or `r/m`, with an optional size). This is to be encoded as an effective address, with a ModR/M byte, an optional SIB byte, and an optional displacement, and the spare (register) field of the ModR/M byte should be the digit given (which will be from 0 to 7, so it fits in three bits). The encoding of effective addresses is given in section A.2.6.

- The code `/r` combines the above two: it indicates that one of the operands is a memory address or `r/m`, and another is a register, and that an effective address should be generated with the spare (register) field in the ModR/M byte being equal to the 'register value' of the register operand. The encoding of effective addresses is given in section A.2.6; register values are given in section A.2.1.

- The codes `ib`, `iw` and `id` indicate that one of the operands to the instruction is an immediate value, and that this is to be encoded as a byte, little-endian word or little-endian doubleword respectively.

- The codes `rb`, `rw` and `rd` indicate that one of the operands to the instruction is an immediate value, and that the *difference* between this value and the address of the end of the instruction is to be encoded as a byte, word or doubleword respectively. Where the form `rw/rd` appears, it indicates that either `rw` or `rd` should be used according to whether assembly is being performed in `BITS 16` or `BITS 32` state respectively.

- The codes `ow` and `od` indicate that one of the operands to the instruction is a reference to the contents of a memory address specified as an immediate value: this encoding is used in some forms of the `MOV` instruction in place of the standard effective-address mechanism. The displacement is encoded as a word or doubleword. Again, `ow/od` denotes that `ow` or `od` should be chosen according to the `BITS` setting.
- The codes `o16` and `o32` indicate that the given form of the instruction should be assembled with operand size 16 or 32 bits. In other words, `o16` indicates a `66` prefix in `BITS 32` state, but generates no code in `BITS 16` state; and `o32` indicates a `66` prefix in `BITS 16` state but generates nothing in `BITS 32`.
- The codes `a16` and `a32`, similarly to `o16` and `o32`, indicate the address size of the given form of the instruction. Where this does not match the `BITS` setting, a `67` prefix is required. Please note that `a16` is useless in long mode as 16-bit addressing is depreciated on the x86-64 architecture extension.

A.2.1 Register Values

Where an instruction requires a register value, it is already implicit in the encoding of the rest of the instruction what type of register is intended: an 8-bit general-purpose register, a segment register, a debug register, an MMX register, or whatever. Therefore there is no problem with registers of different types sharing an encoding value.

Please note that for the register classes listed below, the register extensions (REX) classes require the use of the REX prefix, which is only available when in long mode on an x86-64 processor. This pretty much goes for any register that has a number higher than 7.

The encodings for the various classes of register are:

- 8-bit general registers: `AL` is 0, `CL` is 1, `DL` is 2, `BL` is 3, `AH` is 4, `CH` is 5, `DH` is 6 and `BH` is 7. Please note that `AH`, `BH`, `CH` and `DH` are not addressable when using the REX prefix in long mode.
- 8-bit general register extensions (REX): `SPL` is 4, `BPL` is 5, `SIL` is 6, `DIL` is 7, `R8B` is 8, `R9B` is 9, `R10B` is 10, `R11B` is 11, `R12B` is 12, `R13B` is 13, `R14B` is 14 and `R15B` is 15.
- 16-bit general registers: `AX` is 0, `CX` is 1, `DX` is 2, `BX` is 3, `SP` is 4, `BP` is 5, `SI` is 6, and `DI` is 7.
- 16-bit general register extensions (REX): `R8W` is 8, `R9W` is 9, `R10w` is 10, `R11W` is 11, `R12W` is 12, `R13W` is 13, `R14W` is 14 and `R15W` is 15.
- 32-bit general registers: `EAX` is 0, `ECX` is 1, `EDX` is 2, `EBX` is 3, `ESP` is 4, `EBP` is 5, `ESI` is 6, and `EDI` is 7.
- 32-bit general register extensions (REX): `R8D` is 8, `R9D` is 9, `R10D` is 10, `R11D` is 11, `R12D` is 12, `R13D` is 13, `R14D` is 14 and `R15D` is 15.
- 64-bit general register extensions (REX): `RAX` is 0, `RCX` is 1, `RDX` is 2, `RBX` is 3, `RSP` is 4, `RBP` is 5, `RSI` is 6, `RDI` is 7, `R8` is 8, `R9` is 9, `R10` is 10, `R11` is 11, `R12` is 12, `R13` is 13, `R14` is 14 and `R15` is 15.

- Segment registers: `ES` is 0, `CS` is 1, `SS` is 2, `DS` is 3, `FS` is 4, and `GS` is 5.
- Floating-point registers: `ST0` is 0, `ST1` is 1, `ST2` is 2, `ST3` is 3, `ST4` is 4, `ST5` is 5, `ST6` is 6, and `ST7` is 7.
- 64-bit MMX registers: `MM0` is 0, `MM1` is 1, `MM2` is 2, `MM3` is 3, `MM4` is 4, `MM5` is 5, `MM6` is 6, and `MM7` is 7.
- 128-bit XMM (SSE) registers: `XMM0` is 0, `XMM1` is 1, `XMM2` is 2, `XMM3` is 3, `XMM4` is 4, `XMM5` is 5, `XMM6` is 6 and `XMM7` is 7.
- 128-bit XMM (SSE) register extensions (REX): `XMM8` is 8, `XMM9` is 9, `XMM10` is 10, `XMM11` is 11, `XMM12` is 12, `XMM13` is 13, `XMM14` is 14 and `XMM15` is 15.
- Control registers: `CR0` is 0, `CR2` is 2, `CR3` is 3, and `CR4` is 4.
- Control register extensions: `CR8` is 8.
- Debug registers: `DR0` is 0, `DR1` is 1, `DR2` is 2, `DR3` is 3, `DR6` is 6, and `DR7` is 7.
- Test registers: `TR3` is 3, `TR4` is 4, `TR5` is 5, `TR6` is 6, and `TR7` is 7.

(Note that wherever a register name contains a number, that number is also the register value for that register.)

A.2.2 Condition Codes

The available condition codes are given here, along with their numeric representations as part of opcodes. Many of these condition codes have synonyms, so several will be listed at a time.

In the following descriptions, the word 'either', when applied to two possible trigger conditions, is used to mean 'either or both'. If 'either but not both' is meant, the phrase 'exactly one of' is used.

- `O` is 0 (trigger if the overflow flag is set); `NO` is 1.
- `B`, `C` and `NAE` are 2 (trigger if the carry flag is set); `AE`, `NB` and `NC` are 3.
- `E` and `Z` are 4 (trigger if the zero flag is set); `NE` and `NZ` are 5.
- `BE` and `NA` are 6 (trigger if either of the carry or zero flags is set); `A` and `NBE` are 7.
- `S` is 8 (trigger if the sign flag is set); `NS` is 9.
- `P` and `PE` are 10 (trigger if the parity flag is set); `NP` and `PO` are 11.
- `L` and `NGE` are 12 (trigger if exactly one of the sign and overflow flags is set); `GE` and `NL` are 13.
- `LE` and `NG` are 14 (trigger if either the zero flag is set, or exactly one of the sign and overflow flags is set); `G` and `NLE` are 15.

Note that in all cases, the sense of a condition code may be reversed by changing the low bit of the numeric representation.

For details of when an instruction sets each of the status flags, see the individual instruction, plus the Status Flags reference in section A.2.4

A.2.3 SSE Condition Predicates

The condition predicates for SSE comparison instructions are the codes used as part of the opcode, to determine what form of comparison is being carried out. In each case, the imm8 value is the final byte of the opcode encoding, and the predicate is the code used as part of the mnemonic for the instruction (equivalent to the "cc" in an integer instruction that used a condition code). The instructions that use this will give details of what the various mnemonics are, this table is used to help you work out details of what is happening.

Predicate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Signal Invalid
EQ	000B	equal	A = B		False	No
LT	001B	less-than	A < B		False	Yes
LE	010B	less-than-or-equal	A <= B		False	Yes
---	----	greater than	A > B	Swap Operands, Use LT	False	Yes
---	----	greater-than-or-equal	A >= B	Swap Operands, Use LE	False	Yes
UNORD	011B	unordered	A, B = Unordered		True	No
NEQ	100B	not-equal	A != B		True	No
NLT	101B	not-less-than	NOT(A < B)		True	Yes
NLE	110B	not-less-than-or-equal	NOT(A <= B)		True	Yes
---	----	not-greater than	NOT(A > B)	Swap Operands, Use NLT	True	Yes
---	----	not-greater than-or-equal	NOT(A >= B)	Swap Operands, Use NLE	True	Yes
ORD	111B	ordered	A , B = Ordered		False	No

The unordered relationship is true when at least one of the two values being compared is a NaN or in an unsupported format.

Note that the comparisons which are listed as not having a predicate or encoding can only be achieved through software emulation, as described in the "emulation" column. Note in particular that an instruction such as `greater-than` is not the same as `NLE`, as, unlike with the `CMP` instruction, it has to take into account the possibility of one operand containing a NaN or an unsupported numeric format.

A.2.4 Status Flags

The status flags provide some information about the result of the arithmetic instructions. This information can be used by conditional instructions (such as `Jcc` and `CMOVcc`) as well as by some of the other instructions (such as `ADC` and `INTO`).

There are 6 status flags:

`CF - Carry flag.`

Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

`PF - Parity flag.`

Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

`AF - Adjust flag.`

Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

`ZF - Zero flag.`

Set if the result is zero; cleared otherwise.

`SF - Sign flag.`

Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

`OF - Overflow flag.`

Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

A.2.5 Control Flags

Control Flags control aspects of the CPU's operation. In the 16-bit FLAGS register there are 3 such flags supported on all CPUs down to the 8086. They are:

A.2.5.1 IF - Interrupt flag

If set, enables hardware interrupts (IRQs) to be serviced by the CPU. Note that a MOV or POP to SS (see section A.5.156 and section A.5.244) disables hardware interrupt servicing and Trace flag tracing until after the next instruction, to allow atomic setting of both SS and SP at the same time. (There is a bug that makes this interrupt lockout fail on early 8086/8088 CPUs, which is why 8086-compatible code that sets SS and SP usually also clears the Interrupt flag explicitly first.)

A.2.5.2 DF - Direction flag

If set (called DN ‘Down’), string operations such as MOVSx (section A.5.178) decrement their index registers, (E)SI and/or (E)DI. If clear (called UP), which is the usual default state, the string operations increment their index registers.

A.2.5.3 TF - Trace flag

If set, the CPU will invoke the Trace interrupt (interrupt 1) after the next instruction. There are a few special cases:

- Other interrupt invocations ignore the prior Trace flag status. That is, interrupts are not traced.
- After an interrupt invocation, the IRET (see section A.5.127) restores TF=1, but the Trace interrupt is then only called after another instruction. That is, the Trace interrupt ‘fires too late’.
- When interrupt lockout due to MOV or POP to SS is in effect, no Trace interrupt is invoked.
- When a repeated string operation is invoked, at most one iteration is executed. (E)IP is reset in case more iterations are to be executed.

  If (E)CX was greater than 1, and (for CMPSx and SCASx) after the first comparison the Zero flag is set or clear so as to repeat the operation, the (E)IP address is reset to point to the first prefix opcode and then a Trace interrupt is invoked.

  When (E)CX was 1 prior to the instruction, or the Zero flag after the comparison indicates to break out of the repetition, then one iteration is executed, (E)IP is not reset, and a Trace interrupt is invoked.

  When (E)CX was 0 prior to the instruction, then no iteration is executed, but (E)IP is still incremented to point after the instruction, and a Trace interrupt is invoked.

A.2.6 Effective Address Encoding: ModR/M and SIB

An effective address is encoded in up to three parts: a ModR/M byte, an optional SIB byte, and an optional byte, word or doubleword displacement field.

The ModR/M byte consists of three fields: the `mod` field, ranging from 0 to 3, in the upper two bits of the byte, the `r/m` field, ranging from 0 to 7, in the lower three bits, and the spare (register) field in the middle (bit 3 to bit 5). The spare field is not relevant to the effective address being encoded, and either contains an extension to the instruction opcode or the register value of another operand.

A.2.6.1 ModR/M encoding a register

The ModR/M system can be used to encode a direct register reference rather than a memory access. This is always done by setting the `mod` field to 3 and the `r/m` field to the register value of the register in question (it must be a general-purpose register, and the size of the register must already be implicit in the encoding of the rest of the instruction). In this case, the SIB byte and displacement field are both absent.

A.2.6.2 Memory a16 ModR/M encoding

In 16-bit addressing mode (either `BITS 16` with no `67` prefix, or `BITS 32` with a `67` prefix), the SIB byte is never used. The general rules for `mod` and `r/m` (there is an exception, given below) are:

- The `mod` field gives the length of the displacement field: 0 means no displacement, 1 means one byte, and 2 means two bytes.
- The `r/m` field encodes the combination of registers to be added to the displacement to give the accessed address: 0 means `BX+SI`, 1 means `BX+DI`, 2 means `BP+SI`, 3 means `BP+DI`, 4 means `SI` only, 5 means `DI` only, 6 means `BP` only, and 7 means `BX` only.

However, there is a special case:

- If `mod` is 0 and `r/m` is 6, the effective address encoded is not `[BP]` as the above rules would suggest, but instead `[disp16]`: the displacement field is present and is two bytes long, and no registers are added to the displacement.

Therefore the effective address `[BP]` cannot be encoded as efficiently as `[BX]`; so if you code `[BP]` in a program, NASM adds a notional 8-bit zero displacement, and sets `mod` to 1, `r/m` to 6, and the one-byte displacement field to 0.

If `BP` is used in an address then the default segment register is `SS`. Otherwise, it is `DS`.

A.2.6.3 Memory a32 ModR/M and SIB encoding

In 32-bit addressing mode (either `BITS 16` with a `67` prefix, or `BITS 32` with no `67` prefix) the general rules (again, there are exceptions) for `mod` and `r/m` are:

- The `mod` field gives the length of the displacement field: 0 means no displacement, 1 means one byte, and 2 means four bytes.
- If only one register is to be added to the displacement, and it is not `ESP`, the `r/m` field gives its register value, and the SIB byte is absent. If the `r/m` field is 4 (which would encode `ESP`), the SIB byte is present and gives the combination and scaling of registers to be added to the displacement.

If the SIB byte is present, it describes the combination of registers (an optional base register, and an optional index register scaled by multiplication by 1, 2, 4 or 8) to be added to the displacement. The SIB byte is divided into the `scale` field, in the top two bits, the `index` field in the next three, and the `base` field in the bottom three. The general rules are:

- The `base` field encodes the register value of the base register.
- The `index` field encodes the register value of the index register, unless it is 4, in which case no index register is used (so `ESP` cannot be used as an index register). If a 4 is encoded then the `scale` field is ignored.
- The `scale` field encodes the multiplier by which the index register is scaled before adding it to the base and displacement: 0 encodes a multiplier of 1, 1 encodes 2, 2 encodes 4 and 3 encodes 8.

The exceptions to the 32-bit encoding rules are:

- If `mod` is 0 and `r/m` is 5, the effective address encoded is not `[EBP]` as the above rules would suggest, but instead `[disp32]`: the displacement field is present and is four bytes long, and no registers are added to the displacement.

- If `mod` is 0, `r/m` is 4 (meaning the SIB byte is present) and `base` is 5, the effective address encoded is not `[EBP+index]` as the above rules would suggest, but instead `[disp32+index]`: the displacement field is present and is four bytes long, and there is no base register (but the index register is still processed in the normal way).

- Note that if the prior exception applies, then the `index` register field can also be encoded as 4. This combines the two special cases, resulting in a longer encoding of `[disp32]` without any registers. (This longer encoding differs only in a 64-bit code segment from the short `mod=0 r/m=5` encoding of `[disp32]`.)

If `EBP` or `ESP` is used as the base register in an address then the default segment register is `SS`. Otherwise, it is `DS`. (The use of these registers corresponds to `r/m` = 5 or `base` = 5 or 4, except for the special cases with `mod` = 0.)

A.2.7 Instruction Prefixes

A.2.7.1 8086 Instruction Prefixes

The 8086 instruction prefixes come in three groups:

- Segment overrides
- String instruction repeat
- LOCK

A.2.7.1.1 8086 Segment Overrides

The following opcode bytes are used as 8086 segment override prefixes:

```
ES                              ; 26                     [8086]
CS                              ; 2E                     [8086]
SS                              ; 36                     [8086]
DS                              ; 3E                     [8086]
```

These prefixes can be used to override the segment register used to access an explicit memory operand, or the implicit "source" operand of the following instructions:

- `XLATB`
- `LODS`
- `MOVS` source
- `CMPS` "source" (actually first operand of equivalent `CMP` pseudocode instruction)
- `OUTS`

The implicit memory operands of the following instructions cannot have their segment overridden:

- `STOS`

- SCAS
- MOVS destination
- CMPS "destination" (actually second operand of equivalent CMP pseudocode instruction)
- INS
- PUSH, POP, and more stack accesses. (If an explicit memory operand is used, its segment can still be overridden.)
- Code fetch using CS and (E)IP

A.2.7.1.2 8086 Repeat Prefixes

The following opcode bytes are used as 8086 repeat prefixes:

```
REPE                        ; F3                    [8086]
REPZ                        ; F3                    [8086]
REPNE                       ; F2                    [8086]
REPNZ                       ; F2                    [8086]
REP                         ; F3                    [8086]
```

They can be used to indicate repetition of the string instructions: LODS, STOS, SCAS, MOVS, CMPS, INS, and OUTS. SCAS and CMPS are the only ones for which REPE differs in behaviour from REPNE.

For NASM and lDebug's line assembler, REP is synonymous to REPE. REPZ is a synonym for REPE, and REPNZ likewise for REPNE.

A.2.7.1.3 8086 LOCK Prefix

```
LOCK                        ; F0                    [8086]
```

The lock prefix can be used on Read-Modify-Write instructions that both read and write to the same explicit memory operand. Using the lock prefix on instructions without an RMW memory operand is invalid.

A.2.7.2 286 Instruction Prefixes

The 286 adds 0Fh (originally considered the opcode for POP CS) as a prefix of sorts. It introduces a longer instruction with at least one more opcode byte after the 0Fh prefix. Instructions involving the 0Fh prefix are expanded in this reference with the prefix included.

A.2.7.3 386 Instruction Prefixes

```
FS                          ; 64                    [386]
GS                          ; 65                    [386]
OSIZE                       ; 66                    [386]
ASIZE                       ; 67                    [386]
```

The 386 adds four prefixes.

The two new segment overrides can be used like the 8086 segment overrides. (On the 386 using these prefixes is the only way to access memory using the FS or GS segment registers.)

The OSIZE and ASIZE switch the operand size and address size to the non-default size. In a 16-bit CS they're called O32 and A32, and their presence indicates 32-bit operand size or 32-bit address size. In a 32-bit CS they're conversely called O16 and A16 and indicate 16-bit operand or address size. (Multiple of the same size prefixes do not repeatedly toggle the indicated size. Rather the presence of redundant prefixes is handled just the same as if this prefix occurred only once.)

A.2.8 Register Extensions: The REX Prefix

The Register Extensions, or REX for short, prefix is the means of accessing extended registers on the x86-64 architecture. REX is considered an instruction prefix, but is required to be after all other prefixes and thus immediately before the first instruction opcode itself. So overall, REX can be thought of as an "Opcode Prefix" instead. The REX prefix itself is indicated by a value of 0x4X, where X is one of 16 different combinations of the actual REX flags.

The REX prefix flags consist of four 1-bit extensions fields. These flags are found in the lower nibble of the actual REX prefix opcode. Below is the list of REX prefix flags, from high bit to low bit.

`REX.W`: When set, this flag indicates the use of a 64-bit operand, as opposed to the default of using 32-bit operands as found in 32-bit Protected Mode.

`REX.R`: When set, this flag extends the `reg` (spare) field of the ModR/M byte. Overall, this raises the amount of addressable registers in this field from 8 to 16.

`REX.X`: When set, this flag extends the `index` field of the `SIB` byte. Overall, this raises the amount of addressable registers in this field from 8 to 16.

`REX.B`: When set, this flag extends the `r/m` field of the ModR/M byte. This flag can also represent an extension to the opcode register (`/r`) field. The determination of which is used varies depending on which instruction is used. Overall, this raises the amount of addressable registers in these fields from 8 to 16.

Internal use of the REX prefix by the processor is consistent, yet non-trivial. Most instructions use the REX prefix as indicated by the above flags. Some instructions require the REX prefix to be present even if the flags are empty. Some instructions default to a 64-bit operand and require the REX prefix only for actual register extensions, and thus ignores the `REX.W` field completely.

At any rate, NASM is designed to handle, and fully supports, the REX prefix internally. Please read the appropriate processor documentation for further information on the REX prefix.

You may have noticed that opcodes 0x40 through 0x4F are actually opcodes for the INC/DEC instructions for each General Purpose Register. This is, of course, correct... for legacy x86. While in long mode, opcodes 0x40 through 0x4F are reserved for use as the REX prefix. The other opcode forms of the INC/DEC instructions are used instead.

A.3 Key to Instruction Flags

Given along with each instruction in this appendix is a set of flags, denoting the type of the instruction. The types are as follows:

- `8086`, `186`, `286`, `386`, `486`, `PENT` and `P6` denote the lowest processor type that supports the instruction. Most instructions run on all processors above the given type; those that do not are documented. The Pentium II contains no additional instructions beyond the P6 (Pentium Pro); from the point of view of its instruction set, it can be thought of as a P6 with MMX capability.

- `3DNOW` indicates that the instruction is a 3DNow! one, and will run on the AMD K6-2 and later processors. ATHLON extensions to the 3DNow! instruction set are documented as such.
- `CYRIX` indicates that the instruction is specific to Cyrix processors, for example the extra MMX instructions in the Cyrix extended MMX instruction set.
- `FPU` indicates that the instruction is a floating-point one, and will only run on machines with a coprocessor (automatically including 486DX, Pentium and above).
- `KATMAI` indicates that the instruction was introduced as part of the Katmai New Instruction set. These instructions are available on the Pentium III and later processors. Those which are not specifically SSE instructions are also available on the AMD Athlon.
- `MMX` indicates that the instruction is an MMX one, and will run on MMX-capable Pentium processors and the Pentium II.
- `PRIV` indicates that the instruction is a protected-mode management instruction. Many of these may only be used in protected mode, or only at privilege level zero.
- `SSE` and `SSE2` indicate that the instruction is a Streaming SIMD Extension instruction. These instructions operate on multiple values in a single operation. SSE was introduced with the Pentium III and SSE2 was introduced with the Pentium 4.
- `UNDOC` indicates that the instruction is an undocumented one, and not part of the official Intel Architecture; it may or may not be supported on any given machine.
- `WILLAMETTE` indicates that the instruction was introduced as part of the new instruction set in the Pentium 4 and Intel Xeon processors. These instructions are also known as SSE2 instructions.
- `X64` indicates that the instruction was introduced as part of the new instruction set in the x86-64 architecture extension, commonly referred to as x64, AMD64 or EM64T.

A.4 Emulator notes

A.4.1 Common corner cases

- `POP` to CS is invalid, and used by 286+ CPUs as prefix byte for two-byte instructions. Some 8086s may actually implement it however. It is unclear how this interacts with the prefetch queue, which is known to be purged by every `JMP`, `CALL`, interrupt, or return instruction.
- `LEA` has to encode a memory reference as source operand, and thus not a register.
- `MOV` to and from segregs can encode invalid non-existent segment registers beyond DS, or (on a 386+) beyond GS.
- `MOV` to a segreg mustn't encode CS as destination. Like `POP` to CS, the prefetch queue behaviour is uncertain.
- `HLT` should cause the machine to halt, ideally until the next hardware interrupt occurs. It

may however simply cause the machine to sleep or give up a timeslice in a multitasker. Failing even that is in error.

- SHL and other shift and rotate instructions mask the shift count (either from CL or the immediate operand) with 31 on 186+ machines, except NEC V20/V30. The latter, and 8088/8086, use the entire shift count.
- On the 8088/8086, the divide by zero interrupt points behind the instruction that caused the exception. Otherwise, it points at the instruction.
- AAD and AAM accept immediate bytes other than 10 on some machines, but not on the NEC V20/V30.
- Interrupt 6 (Invalid opcode) didn't exist on the 8088/8086. Unsupported instructions may behave as one- or two-byte no-ops or otherwise oddly.
- Interrupt lockout after MOV or POP to SS is described in the Control Flags description of the Interrupt Flag, section A.2.5.1.
- Proper repeated string operations tracing is described in the Control Flags description of the Trace Flag, section A.2.5.3.

A.4.2 Emulator call encodings

- NTVDM uses C4 C4 (LES with a register source) followed by one or two callback identifier bytes.
- DOSBox uses FE 38 (invalid encoding with /7) followed by a callback identifier word.
- dosemu2 uses HLT and dispatches based on the address of the HLT.
- 8086tiny uses 0F followed by one identifier byte, currently in the range 00 to 04.

A.5 x86 Instruction Set

A.5.1 AAA, AAS, AAM, AAD: ASCII Adjustments

```
AAA                              ; 37                [8086]

AAS                              ; 3F                [8086]

AAD                              ; D5 0A             [8086]
AAD imm8                         ; D5 ib             [8086]

AAM                              ; D4 0A             [8086]
AAM imm8                         ; D4 ib             [8086]
```

These instructions are used in conjunction with the add, subtract, multiply and divide instructions to perform binary-coded decimal arithmetic in *unpacked* (one BCD digit per byte - easy to translate to and from ASCII, hence the instruction names) form. There are also packed BCD instructions DAA and DAS: see section A.5.57.

- AAA (ASCII Adjust After Addition) should be used after a one-byte ADD instruction whose destination was the AL register: by means of examining the value in the low nibble of AL and also the auxiliary carry flag AF, it determines whether the addition has overflowed, and adjusts it (and sets the carry flag) if so. You can add long BCD strings together by doing ADD/AAA on the low digits, then doing ADC/AAA on each subsequent digit.

- AAS (ASCII Adjust AL After Subtraction) works similarly to AAA, but is for use after SUB instructions rather than ADD.
- AAM (ASCII Adjust AX After Multiply) is for use after you have multiplied two decimal digits together and left the result in AL: it divides AL by ten and stores the quotient in AH, leaving the remainder in AL. The divisor 10 can be changed by specifying an operand to the instruction: a particularly handy use of this is AAM 16, causing the two nibbles in AL to be separated into AH and AL. Note that the divisor immediate byte is ignored by some implementations such as the NEC V20/V30, which always use 10 as divisor.
- AAD (ASCII Adjust AX Before Division) performs the inverse operation to AAM: it multiplies AH by ten, adds it to AL, and sets AH to zero. Again, the multiplier 10 can be changed on some implementations.

A.5.2 ADC: Add with Carry

```
ADC r/m8,reg8                  ; 10 /r                 [8086]
ADC r/m16,reg16                ; o16 11 /r             [8086]
ADC r/m32,reg32                ; o32 11 /r             [386]

ADC reg8,r/m8                  ; 12 /r                 [8086]
ADC reg16,r/m16                ; o16 13 /r             [8086]
ADC reg32,r/m32                ; o32 13 /r             [386]

ADC r/m8,imm8                  ; 80 /2 ib              [8086]
ADC r/m16,imm16                ; o16 81 /2 iw          [8086]
ADC r/m32,imm32                ; o32 81 /2 id          [386]

ADC r/m16,imm8                 ; o16 83 /2 ib          [8086]
ADC r/m32,imm8                 ; o32 83 /2 ib          [386]

ADC AL,imm8                    ; 14 ib                 [8086]
ADC AX,imm16                   ; o16 15 iw             [8086]
ADC EAX,imm32                  ; o32 15 id             [386]
```

ADC performs integer addition: it adds its two operands together, plus the value of the carry flag, and leaves the result in its destination (first) operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.

The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent ADC instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The BYTE qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the STRICT WORD or STRICT DWORD qualifier.

To add two numbers without also adding the contents of the carry flag, use ADD (section A.5.3).

A.5.3 ADD: Add Integers

```
ADD r/m8,reg8                  ; 00 /r                [8086]
ADD r/m16,reg16                ; o16 01 /r            [8086]
ADD r/m32,reg32                ; o32 01 /r            [386]

ADD reg8,r/m8                  ; 02 /r                [8086]
ADD reg16,r/m16                ; o16 03 /r            [8086]
ADD reg32,r/m32                ; o32 03 /r            [386]

ADD r/m8,imm8                  ; 80 /7 ib             [8086]
ADD r/m16,imm16                ; o16 81 /7 iw         [8086]
ADD r/m32,imm32                ; o32 81 /7 id         [386]

ADD r/m16,imm8                 ; o16 83 /7 ib         [8086]
ADD r/m32,imm8                 ; o32 83 /7 ib         [386]

ADD AL,imm8                    ; 04 ib                [8086]
ADD AX,imm16                   ; o16 05 iw            [8086]
ADD EAX,imm32                  ; o32 05 id            [386]
```

`ADD` performs integer addition: it adds its two operands together, and leaves the result in its destination (first) operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.

The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `ADC` instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The `BYTE` qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the `STRICT WORD` or `STRICT DWORD` qualifier.

A.5.4 ADDPD: ADD Packed Double-Precision FP Values

```
ADDPD xmm1,xmm2/mem128         ; 66 0F 58 /r     [WILLAMETTE,SSE2]
```

`ADDPD` performs addition on each of two packed double-precision FP value pairs.

```
   dst[0-63]   := dst[0-63]   + src[0-63],
   dst[64-127] := dst[64-127] + src[64-127].
```

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

A.5.5 ADDPS: ADD Packed Single-Precision FP Values

```
ADDPS xmm1,xmm2/mem128         ; 0F 58 /r        [KATMAI,SSE]
```

`ADDPS` performs addition on each of four packed single-precision FP value pairs

```
   dst[0-31]   := dst[0-31]   + src[0-31],
   dst[32-63]  := dst[32-63]  + src[32-63],
```

```
dst[64-95]  := dst[64-95]  + src[64-95],
dst[96-127] := dst[96-127] + src[96-127].
```

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

A.5.6 ADDSD: ADD Scalar Double-Precision FP Values

```
ADDSD xmm1,xmm2/mem64          ; F2 0F 58 /r     [KATMAI,SSE]
```

ADDSD adds the low double-precision FP values from the source and destination operands and stores the double-precision FP result in the destination operand.

```
dst[0-63]   := dst[0-63] + src[0-63],
dst[64-127] remains unchanged.
```

The destination is an XMM register. The source operand can be either an XMM register or a 64-bit memory location.

A.5.7 ADDSS: ADD Scalar Single-Precision FP Values

```
ADDSS xmm1,xmm2/mem32          ; F3 0F 58 /r     [WILLAMETTE,SSE2]
```

ADDSS adds the low single-precision FP values from the source and destination operands and stores the single-precision FP result in the destination operand.

```
dst[0-31]   := dst[0-31] + src[0-31],
dst[32-127] remains unchanged.
```

The destination is an XMM register. The source operand can be either an XMM register or a 32-bit memory location.

A.5.8 AND: Bitwise AND

```
AND r/m8,reg8                  ; 20 /r                [8086]
AND r/m16,reg16                ; o16 21 /r            [8086]
AND r/m32,reg32                ; o32 21 /r            [386]

AND reg8,r/m8                  ; 22 /r                [8086]
AND reg16,r/m16                ; o16 23 /r            [8086]
AND reg32,r/m32                ; o32 23 /r            [386]

AND r/m8,imm8                  ; 80 /4 ib             [8086]
AND r/m16,imm16                ; o16 81 /4 iw         [8086]
AND r/m32,imm32                ; o32 81 /4 id         [386]

AND r/m16,imm8                 ; o16 83 /4 ib         [8086]
AND r/m32,imm8                 ; o32 83 /4 ib         [386]

AND AL,imm8                    ; 24 ib                [8086]
AND AX,imm16                   ; o16 25 iw            [8086]
AND EAX,imm32                  ; o32 25 id            [386]
```

AND performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The `BYTE` qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the `STRICT WORD` or `STRICT DWORD` qualifier.

The Carry Flag is cleared by `AND`. The Zero Flag is set according to whether the result is zero.

The `TEST` (see section A.5.317) instruction performs the same operation as `AND` but without writing to the destination operand. That is, it only reads the operands and writes the status flags.

The `MMX` instruction `PAND` (see section A.5.202) performs the same operation on the 64-bit `MMX` registers.

A.5.9 `ANDNPD`: Bitwise Logical AND NOT of Packed Double-Precision FP Values

```
ANDNPD xmm1,xmm2/mem128        ; 66 0F 55 /r     [WILLAMETTE,SSE2]
```

`ANDNPD` inverts the bits of the two double-precision floating-point values in the destination register, and then performs a logical AND between the two double-precision floating-point values in the source operand and the temporary inverted result, storing the result in the destination register.

```
dst[0-63]   := src[0-63]   AND NOT dst[0-63],
dst[64-127] := src[64-127] AND NOT dst[64-127].
```

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

A.5.10 `ANDNPS`: Bitwise Logical AND NOT of Packed Single-Precision FP Values

```
ANDNPS xmm1,xmm2/mem128        ; 0F 55 /r        [KATMAI,SSE]
```

`ANDNPS` inverts the bits of the four single-precision floating-point values in the destination register, and then performs a logical AND between the four single-precision floating-point values in the source operand and the temporary inverted result, storing the result in the destination register.

```
dst[0-31]   := src[0-31]   AND NOT dst[0-31],
dst[32-63]  := src[32-63]  AND NOT dst[32-63],
dst[64-95]  := src[64-95]  AND NOT dst[64-95],
dst[96-127] := src[96-127] AND NOT dst[96-127].
```

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

A.5.11 `ANDPD`: Bitwise Logical AND For Single FP

```
ANDPD xmm1,xmm2/mem128         ; 66 0F 54 /r     [WILLAMETTE,SSE2]
```

`ANDPD` performs a bitwise logical AND of the two double-precision floating point values in the source and destination operand, and stores the result in the destination register.

```
dst[0-63]   := src[0-63]   AND dst[0-63],
dst[64-127] := src[64-127] AND dst[64-127].
```

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

A.5.12 `ANDPS`: Bitwise Logical AND For Single FP

```
ANDPS xmm1,xmm2/mem128        ; 0F 54 /r        [KATMAI,SSE]
```

`ANDPS` performs a bitwise logical AND of the four single-precision floating point values in the source and destination operand, and stores the result in the destination register.

```
dst[0-31]   := src[0-31]   AND dst[0-31],
dst[32-63]  := src[32-63]  AND dst[32-63],
dst[64-95]  := src[64-95]  AND dst[64-95],
dst[96-127] := src[96-127] AND dst[96-127].
```

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

A.5.13 `ARPL`: Adjust RPL Field of Selector

```
ARPL r/m16,reg16              ; 63 /r                [286,PRIV]
```

`ARPL` expects its two word operands to be segment selectors. It adjusts the `RPL` (requested privilege level - stored in the bottom two bits of the selector) field of the destination (first) operand to ensure that it is no less (i.e. no more privileged than) the `RPL` field of the source operand. The zero flag is set if and only if a change had to be made.

A.5.14 `BOUND`: Check Array Index against Bounds

```
BOUND reg16,mem               ; o16 62 /r            [186]
BOUND reg32,mem               ; o32 62 /r            [386]
```

`BOUND` expects its second operand to point to an area of memory containing two signed values of the same size as its first operand (i.e. two words for the 16-bit form; two doublewords for the 32-bit form). It performs two signed comparisons: if the value in the register passed as its first operand is less than the first of the in-memory values, or is greater than or equal to the second, it throws a `BR` exception. Otherwise, it does nothing.

A.5.15 `BSF`, `BSR`: Bit Scan

```
BSF reg16,r/m16               ; o16 0F BC /r         [386]
BSF reg32,r/m32               ; o32 0F BC /r         [386]

BSR reg16,r/m16               ; o16 0F BD /r         [386]
BSR reg32,r/m32               ; o32 0F BD /r         [386]
```

- `BSF` searches for the least significant set bit in its source (second) operand, and if it finds one, stores the index in its destination (first) operand. If no set bit is found, the contents of the destination operand are undefined. If the source operand is zero, the zero flag is set.

- BSR performs the same function, but searches from the top instead, so it finds the most significant set bit.

Bit indices are from 0 (least significant) to 15 or 31 (most significant). The destination operand can only be a register. The source operand can be a register or a memory location.

A.5.16 BSWAP: Byte Swap

```
BSWAP reg32                   ; o32 0F C8+r           [486]
```

BSWAP swaps the order of the four bytes of a 32-bit register: bits 0-7 exchange places with bits 24-31, and bits 8-15 swap with bits 16-23. There is no explicit 16-bit equivalent: to byte-swap AX, BX, CX or DX, XCHG can be used. When BSWAP is used with a 16-bit register, the result is undefined.

A.5.17 BT, BTC, BTR, BTS: Bit Test

```
BT r/m16,reg16                ; o16 0F A3 /r          [386]
BT r/m32,reg32                ; o32 0F A3 /r          [386]
BT r/m16,imm8                 ; o16 0F BA /4 ib       [386]
BT r/m32,imm8                 ; o32 0F BA /4 ib       [386]

BTC r/m16,reg16               ; o16 0F BB /r          [386]
BTC r/m32,reg32               ; o32 0F BB /r          [386]
BTC r/m16,imm8                ; o16 0F BA /7 ib       [386]
BTC r/m32,imm8                ; o32 0F BA /7 ib       [386]

BTR r/m16,reg16               ; o16 0F B3 /r          [386]
BTR r/m32,reg32               ; o32 0F B3 /r          [386]
BTR r/m16,imm8                ; o16 0F BA /6 ib       [386]
BTR r/m32,imm8                ; o32 0F BA /6 ib       [386]

BTS r/m16,reg16               ; o16 0F AB /r          [386]
BTS r/m32,reg32               ; o32 0F AB /r          [386]
BTS r/m16,imm8                ; o16 0F BA /5 ib       [386]
BTS r/m32,imm8                ; o32 0F BA /5 ib       [386]
```

These instructions all test one bit of their first operand, whose index is given by the second operand, and store the value of that bit into the carry flag. Bit indices are from 0 (least significant) to 15 or 31 (most significant).

In addition to storing the original value of the bit into the carry flag, BTR also resets (clears) the bit in the operand itself. BTS sets the bit, and BTC complements the bit. BT does not modify its operands.

The destination can be a register or a memory location. The source can be a register or an immediate value.

If the destination operand is a register, the bit offset should be in the range 0-15 (for 16-bit operands) or 0-31 (for 32-bit operands). An immediate value outside these ranges will be taken modulo 16/32 by the processor.

If the destination operand is a memory location, then an immediate bit offset follows the same rules as for a register. If the bit offset is in a register, then it can be anything within the signed range of the register used (ie, for a 32-bit operand, it can be (-2^31) to (2^31 - 1)).

A.5.18 `CALL`: Call Subroutine

```
CALL imm                      ; E8 rw/rd                [8086]
CALL imm:imm16                ; o16 9A iw iw            [8086]
CALL imm:imm32                ; o32 9A id iw            [386]
CALL FAR mem16                ; o16 FF /3               [8086]
CALL FAR mem32                ; o32 FF /3               [386]
CALL r/m16                    ; o16 FF /2               [8086]
CALL r/m32                    ; o32 FF /2               [386]
```

`CALL` calls a subroutine, by means of pushing the current instruction pointer (`IP`) and optionally `CS` as well on the stack, and then jumping to a given address.

`CS` is pushed as well as `IP` if and only if the call is a far call, i.e. a destination segment address is specified in the instruction. The forms involving two colon-separated arguments are far calls; so are the `CALL FAR mem` forms.

The immediate near call takes one of two forms (`CALL imm16/imm32`), determined by the current segment size limit. For 16-bit operands, you would use `CALL 0x1234`, and for 32-bit operands you would use `CALL 0x12345678`. The value passed as an operand is a relative offset.

You can choose between the two immediate far call forms (`CALL imm:imm`) by the use of the `WORD` and `DWORD` keywords: `CALL WORD 0x1234:0x5678` or `CALL DWORD 0x1234:0x56789abc`.

The `CALL FAR mem` forms execute a far call by loading the destination address out of memory. The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using `CALL WORD FAR mem` or `CALL DWORD FAR mem`.

The `CALL r/m` forms execute a near call (within the same segment), loading the destination address out of memory or out of a register. The keyword `NEAR` may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using `CALL WORD mem` or `CALL DWORD mem`.

As a convenience, NASM does not require you to call a far procedure symbol by coding the cumbersome `CALL SEG routine:routine`, but instead allows the easier synonym `CALL FAR routine`.

A.5.19 `CBW`, `CWD`, `CDQ`, `CWDE`: Sign Extensions

```
CBW                           ; o16 98                  [8086]
CWDE                          ; o32 98                  [386]

CWD                           ; o16 99                  [8086]
CDQ                           ; o32 99                  [386]
```

All these instructions sign-extend a short value into a longer one, by replicating the top bit of the original value to fill the extended one.

`CBW` extends `AL` into `AX` by repeating the top bit of `AL` in every bit of `AH`. `CWDE` extends `AX` into `EAX`. `CWD` extends `AX` into `DX:AX` by repeating the top bit of `AX` throughout `DX`, and `CDQ` extends `EAX` into `EDX:EAX`.

A.5.20 CLC, CLD, CLI, CLTS: Clear Flags

```
CLC                              ; F8                     [8086]
CLD                              ; FC                     [8086]
CLI                              ; FA                     [8086]
CLTS                             ; 0F 06                  [286,PRIV]
```

These instructions clear various flags. CLC clears the carry flag; CLD clears the direction flag; CLI clears the interrupt flag (thus disabling interrupts); and CLTS clears the task-switched (TS) flag in CR0.

To set the carry, direction, or interrupt flags, use the STC, STD and STI instructions (section A.5.301). To invert the carry flag, use CMC (section A.5.22).

A.5.21 CLFLUSH: Flush Cache Line

```
CLFLUSH mem                      ; 0F AE /7          [WILLAMETTE,SSE2]
```

CLFLUSH invalidates the cache line that contains the linear address specified by the source operand from all levels of the processor cache hierarchy (data and instruction). 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 points to a byte-sized memory location.

Although CLFLUSH is flagged SSE2 and above, it may not be present on all processors which have SSE2 support, and it may be supported on other processors; the CPUID instruction (section A.5.34) will return a bit which indicates support for the CLFLUSH instruction.

A.5.22 CMC: Complement Carry Flag

```
CMC                              ; F5                     [8086]
```

CMC changes the value of the carry flag: if it was 0, it sets it to 1, and vice versa.

A.5.23 CMOVcc: Conditional Move

```
CMOVcc reg16,r/m16               ; o16 0F 40+cc /r        [P6]
CMOVcc reg32,r/m32               ; o32 0F 40+cc /r        [P6]
```

CMOV moves its source (second) operand into its destination (first) operand if the given condition code is satisfied; otherwise it does nothing.

For a list of condition codes, see section A.2.2.

Although the CMOV instructions are flagged P6 and above, they may not be supported by all Pentium Pro processors; the CPUID instruction (section A.5.34) will return a bit which indicates whether conditional moves are supported.

A.5.24 CMP: Compare Integers

```
CMP r/m8,reg8                    ; 38 /r                  [8086]
CMP r/m16,reg16                  ; o16 39 /r              [8086]
CMP r/m32,reg32                  ; o32 39 /r              [386]

CMP reg8,r/m8                    ; 3A /r                  [8086]
CMP reg16,r/m16                  ; o16 3B /r              [8086]
```

```
CMP reg32,r/m32                ; o32 3B /r            [386]

CMP r/m8,imm8                  ; 80 /7 ib             [8086]
CMP r/m16,imm16                ; o16 81 /7 iw         [8086]
CMP r/m32,imm32                ; o32 81 /7 id         [386]

CMP r/m16,imm8                 ; o16 83 /7 ib         [8086]
CMP r/m32,imm8                 ; o32 83 /7 ib         [386]

CMP AL,imm8                    ; 3C ib                [8086]
CMP AX,imm16                   ; o16 3D iw            [8086]
CMP EAX,imm32                  ; o32 3D id            [386]
```

`CMP` performs a ‘mental’ subtraction of its second operand from its first operand, and affects the flags as if the subtraction had taken place, but does not store the result of the subtraction anywhere. (For subtraction that does store the result, see section A.5.305.)

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The `BYTE` qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the `STRICT WORD` or `STRICT DWORD` qualifier.

The destination operand can be a register or a memory location. The source can be a register, memory location or an immediate value of the same size as the destination.

A.5.25 `CMPccPD`: Packed Double-Precision FP Compare

```
CMPPD xmm1,xmm2/mem128,imm8   ; 66 0F C2 /r ib  [WILLAMETTE,SSE2]

CMPEQPD xmm1,xmm2/mem128      ; 66 0F C2 /r 00  [WILLAMETTE,SSE2]
CMPLTPD xmm1,xmm2/mem128      ; 66 0F C2 /r 01  [WILLAMETTE,SSE2]
CMPLEPD xmm1,xmm2/mem128      ; 66 0F C2 /r 02  [WILLAMETTE,SSE2]
CMPUNORDPD xmm1,xmm2/mem128   ; 66 0F C2 /r 03  [WILLAMETTE,SSE2]
CMPNEQPD xmm1,xmm2/mem128     ; 66 0F C2 /r 04  [WILLAMETTE,SSE2]
CMPNLTPD xmm1,xmm2/mem128     ; 66 0F C2 /r 05  [WILLAMETTE,SSE2]
CMPNLEPD xmm1,xmm2/mem128     ; 66 0F C2 /r 06  [WILLAMETTE,SSE2]
CMPORDPD xmm1,xmm2/mem128     ; 66 0F C2 /r 07  [WILLAMETTE,SSE2]
```

The `CMPccPD` instructions compare the two packed double-precision FP values in the source and destination operands, and returns the result of the comparison in the destination register. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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

The third operand is an 8-bit immediate value, of which the low 3 bits define the type of comparison. For ease of programming, the 8 two-operand pseudo-instructions are provided, with the third operand already filled in. The `Condition Predicates` are:

```
EQ     0   Equal
LT     1   Less-than
```

```
LE      2   Less-than-or-equal
UNORD   3   Unordered
NE      4   Not-equal
NLT     5   Not-less-than
NLE     6   Not-less-than-or-equal
ORD     7   Ordered
```

For more details of the comparison predicates, and details of how to emulate the "greater-than" equivalents, see section A.2.3

A.5.26 CMPccPS: Packed Single-Precision FP Compare

```
CMPPS xmm1,xmm2/mem128,imm8    ; 0F C2 /r ib     [KATMAI,SSE]

CMPEQPS xmm1,xmm2/mem128       ; 0F C2 /r 00     [KATMAI,SSE]
CMPLTPS xmm1,xmm2/mem128       ; 0F C2 /r 01     [KATMAI,SSE]
CMPLEPS xmm1,xmm2/mem128       ; 0F C2 /r 02     [KATMAI,SSE]
CMPUNORDPS xmm1,xmm2/mem128    ; 0F C2 /r 03     [KATMAI,SSE]
CMPNEQPS xmm1,xmm2/mem128      ; 0F C2 /r 04     [KATMAI,SSE]
CMPNLTPS xmm1,xmm2/mem128      ; 0F C2 /r 05     [KATMAI,SSE]
CMPNLEPS xmm1,xmm2/mem128      ; 0F C2 /r 06     [KATMAI,SSE]
CMPORDPS xmm1,xmm2/mem128      ; 0F C2 /r 07     [KATMAI,SSE]
```

The `CMPccPS` instructions compare the two packed single-precision FP values in the source and destination operands, and returns the result of the comparison in the destination register. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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

The third operand is an 8-bit immediate value, of which the low 3 bits define the type of comparison. For ease of programming, the 8 two-operand pseudo-instructions are provided, with the third operand already filled in. The `Condition Predicates` are:

```
EQ      0   Equal
LT      1   Less-than
LE      2   Less-than-or-equal
UNORD   3   Unordered
NE      4   Not-equal
NLT     5   Not-less-than
NLE     6   Not-less-than-or-equal
ORD     7   Ordered
```

For more details of the comparison predicates, and details of how to emulate the "greater-than" equivalents, see section A.2.3

A.5.27 CMPSB, CMPSW, CMPSD: Compare Strings

```
CMPSB                          ; A6                   [8086]
CMPSW                          ; o16 A7               [8086]
CMPSD                          ; o32 A7               [386]
```

`CMPSB` compares the byte at `[DS:SI]` or `[DS:ESI]` with the byte at `[ES:DI]` or

[ES:EDI], and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) SI and DI (or ESI and EDI).

The registers used are SI and DI if the address size is 16 bits, and ESI and EDI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

The segment register used to load from [SI] or [ESI] can be overridden by using a segment register name as a prefix (for example, ES CMPSB). The use of ES for the load from [DI] or [EDI] cannot be overridden.

CMPSW and CMPSD work in the same way, but they compare a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REPE and REPNE prefixes (equivalently, REPZ and REPNZ) may be used to repeat the instruction up to CX (or ECX - again, the address size chooses which) times until the first unequal or equal element is found. To NASM, REP is an alias for REPE.

Note that CMPSB is equivalent to LODSB followed by SCASB, except it doesn't write the accumulator register (AL).

A.5.27.1 Pseudo-code examples

a16 CMPSB without segment override and with Direction Flag clear (UP) is equal to

```
CMP BYTE [SI], BYTE [ES:DI]
LEA SI, [SI + 1]
LEA DI, [DI + 1]
```

a16 REPE CMPSW without segment override and with Direction Flag clear (UP) is equal to

```
JCXZ @FF
@@:
CMP WORD [SI], WORD [ES:DI]
LEA SI, [SI + 2]
LEA DI, [DI + 2]
a16 LOOPE @B
@@:
```

a32 ES CMPSD with Direction Flag set (DN) is equal to

```
CMP DWORD [ES:ESI], DWORD [ES:EDI]
LEA ESI, [ESI - 4]
LEA EDI, [EDI - 4]
```

A.5.28 CMPccSD: Scalar Double-Precision FP Compare

```
CMPSD xmm1,xmm2/mem64,imm8    ; F2 0F C2 /r ib  [WILLAMETTE,SSE2]

CMPEQSD xmm1,xmm2/mem64       ; F2 0F C2 /r 00  [WILLAMETTE,SSE2]
CMPLTSD xmm1,xmm2/mem64       ; F2 0F C2 /r 01  [WILLAMETTE,SSE2]
CMPLESD xmm1,xmm2/mem64       ; F2 0F C2 /r 02  [WILLAMETTE,SSE2]
CMPUNORDSD xmm1,xmm2/mem64    ; F2 0F C2 /r 03  [WILLAMETTE,SSE2]
```

```
CMPNEQSD xmm1,xmm2/mem64       ; F2 0F C2 /r 04  [WILLAMETTE,SSE2]
CMPNLTSD xmm1,xmm2/mem64       ; F2 0F C2 /r 05  [WILLAMETTE,SSE2]
CMPNLESD xmm1,xmm2/mem64       ; F2 0F C2 /r 06  [WILLAMETTE,SSE2]
CMPORDSD xmm1,xmm2/mem64       ; F2 0F C2 /r 07  [WILLAMETTE,SSE2]
```

The `CMPccSD` instructions compare the low-order double-precision FP values in the source and destination operands, and returns the result of the comparison in the destination register. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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

The third operand is an 8-bit immediate value, of which the low 3 bits define the type of comparison. For ease of programming, the 8 two-operand pseudo-instructions are provided, with the third operand already filled in. The `Condition Predicates` are:

```
EQ     0   Equal
LT     1   Less-than
LE     2   Less-than-or-equal
UNORD  3   Unordered
NE     4   Not-equal
NLT    5   Not-less-than
NLE    6   Not-less-than-or-equal
ORD    7   Ordered
```

For more details of the comparison predicates, and details of how to emulate the "greater-than" equivalents, see section A.2.3

A.5.29 CMPccSS: Scalar Single-Precision FP Compare

```
CMPSS xmm1,xmm2/mem32,imm8     ; F3 0F C2 /r ib  [KATMAI,SSE]

CMPEQSS xmm1,xmm2/mem32        ; F3 0F C2 /r 00  [KATMAI,SSE]
CMPLTSS xmm1,xmm2/mem32        ; F3 0F C2 /r 01  [KATMAI,SSE]
CMPLESS xmm1,xmm2/mem32        ; F3 0F C2 /r 02  [KATMAI,SSE]
CMPUNORDSS xmm1,xmm2/mem32     ; F3 0F C2 /r 03  [KATMAI,SSE]
CMPNEQSS xmm1,xmm2/mem32       ; F3 0F C2 /r 04  [KATMAI,SSE]
CMPNLTSS xmm1,xmm2/mem32       ; F3 0F C2 /r 05  [KATMAI,SSE]
CMPNLESS xmm1,xmm2/mem32       ; F3 0F C2 /r 06  [KATMAI,SSE]
CMPORDSS xmm1,xmm2/mem32       ; F3 0F C2 /r 07  [KATMAI,SSE]
```

The `CMPccSS` instructions compare the low-order single-precision FP values in the source and destination operands, and returns the result of the comparison in the destination register. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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

The third operand is an 8-bit immediate value, of which the low 3 bits define the type of comparison. For ease of programming, the 8 two-operand pseudo-instructions are provided, with the third operand already filled in. The `Condition Predicates` are:

```
EQ     0   Equal
LT     1   Less-than
LE     2   Less-than-or-equal
UNORD  3   Unordered
NE     4   Not-equal
NLT    5   Not-less-than
NLE    6   Not-less-than-or-equal
ORD    7   Ordered
```

For more details of the comparison predicates, and details of how to emulate the "greater-than" equivalents, see section A.2.3

A.5.30 `CMPXCHG`, `CMPXCHG486`: Compare and Exchange

```
CMPXCHG r/m8,reg8              ; 0F B0 /r             [PENT]
CMPXCHG r/m16,reg16            ; o16 0F B1 /r         [PENT]
CMPXCHG r/m32,reg32            ; o32 0F B1 /r         [PENT]

CMPXCHG486 r/m8,reg8           ; 0F A6 /r             [486,UNDOC]
CMPXCHG486 r/m16,reg16         ; o16 0F A7 /r         [486,UNDOC]
CMPXCHG486 r/m32,reg32         ; o32 0F A7 /r         [486,UNDOC]
```

These two instructions perform exactly the same operation; however, apparently some (not all) 486 processors support it under a non-standard opcode, so NASM provides the undocumented `CMPXCHG486` form to generate the non-standard opcode.

`CMPXCHG` compares its destination (first) operand to the value in `AL`, `AX` or `EAX` (depending on the operand size of the instruction). If they are equal, it copies its source (second) operand into the destination and sets the zero flag. Otherwise, it clears the zero flag and copies the destination register to AL, AX or EAX.

The destination can be either a register or a memory location. The source is a register.

`CMPXCHG` is intended to be used for atomic operations in multitasking or multiprocessor environments. To safely update a value in shared memory, for example, you might load the value into `EAX`, load the updated value into `EBX`, and then execute the instruction `LOCK CMPXCHG [value],EBX`. If `value` has not changed since being loaded, it is updated with your desired new value, and the zero flag is set to let you know it has worked. (The `LOCK` prefix prevents another processor doing anything in the middle of this operation: it guarantees atomicity.) However, if another processor has modified the value in between your load and your attempted store, the store does not happen, and you are notified of the failure by a cleared zero flag, so you can go round and try again.

A.5.31 `CMPXCHG8B`: Compare and Exchange Eight Bytes

```
CMPXCHG8B mem                  ; 0F C7 /1             [PENT]
```

This is a larger and more unwieldy version of `CMPXCHG`: it compares the 64-bit (eight-byte) value stored at `[mem]` with the value in `EDX:EAX`. If they are equal, it sets the zero flag and stores `ECX:EBX` into the memory area. If they are unequal, it clears the zero flag and stores the memory contents into `EDX:EAX`.

`CMPXCHG8B` can be used with the `LOCK` prefix, to allow atomic execution. This is useful in multi-processor and multi-tasking environments.

A.5.32 `COMISD`: Scalar Ordered Double-Precision FP Compare and Set EFLAGS

```
COMISD xmm1,xmm2/mem64          ; 66 0F 2F /r     [WILLAMETTE,SSE2]
```

`COMISD` compares the low-order double-precision FP value in the two source operands. ZF, PF and CF are set according to the result. OF, AF and AF are cleared. The unordered result is returned if either source is a NaN (QNaN or SNaN).

The destination operand is an `XMM` register. The source can be either an `XMM` register or a memory location.

The flags are set according to the following rules:

```
   Result           Flags        Values

   UNORDERED:       ZF,PF,CF <-- 111;
   GREATER_THAN:    ZF,PF,CF <-- 000;
   LESS_THAN:       ZF,PF,CF <-- 001;
   EQUAL:           ZF,PF,CF <-- 100;
```

A.5.33 `COMISS`: Scalar Ordered Single-Precision FP Compare and Set EFLAGS

```
COMISS xmm1,xmm2/mem32          ; 66 0F 2F /r     [KATMAI,SSE]
```

`COMISS` compares the low-order single-precision FP value in the two source operands. ZF, PF and CF are set according to the result. OF, AF and AF are cleared. The unordered result is returned if either source is a NaN (QNaN or SNaN).

The destination operand is an `XMM` register. The source can be either an `XMM` register or a memory location.

The flags are set according to the following rules:

```
   Result           Flags        Values

   UNORDERED:       ZF,PF,CF <-- 111;
   GREATER_THAN:    ZF,PF,CF <-- 000;
   LESS_THAN:       ZF,PF,CF <-- 001;
   EQUAL:           ZF,PF,CF <-- 100;
```

A.5.34 `CPUID`: Get CPU Identification Code

```
CPUID                           ; 0F A2                   [PENT]
```

`CPUID` returns various information about the processor it is being executed on. It fills the four registers `EAX`, `EBX`, `ECX` and `EDX` with information, which varies depending on the input contents of `EAX`.

`CPUID` also acts as a barrier to serialize instruction execution: executing the `CPUID` instruction guarantees that all the effects (memory modification, flag modification, register modification) of previous instructions have been completed before the next instruction gets fetched.

The information returned is as follows:

- If EAX is zero on input, EAX on output holds the maximum acceptable input value of EAX, and EBX:EDX:ECX contain the string 'GenuineIntel' (or not, if you have a clone processor). That is to say, EBX contains 'Genu' (in NASM's own sense of character constants), EDX contains 'ineI' and ECX contains 'ntel'.
- If EAX is one on input, EAX on output contains version information about the processor, and EDX contains a set of feature flags, showing the presence and absence of various features. For example, bit 8 is set if the CMPXCHG8B instruction (section A.5.31) is supported, bit 15 is set if the conditional move instructions (section A.5.23 and section A.5.72) are supported, and bit 23 is set if MMX instructions are supported.
- If EAX is two on input, EAX, EBX, ECX and EDX all contain information about caches and TLBs (Translation Lookahead Buffers).

For more information on the data returned from CPUID, see the documentation from Intel and other processor manufacturers.

A.5.35 CVTDQ2PD: Packed Signed INT32 to Packed Double-Precision FP Conversion

```
CVTDQ2PD xmm1,xmm2/mem64      ; F3 0F E6 /r     [WILLAMETTE,SSE2]
```

CVTDQ2PD converts two packed signed doublewords from the source operand to two packed double-precision FP values in the destination operand.

The destination operand is an XMM register. The source can be either an XMM register or a 64-bit memory location. If the source is a register, the packed integers are in the low quadword.

A.5.36 CVTDQ2PS: Packed Signed INT32 to Packed Single-Precision FP Conversion

```
CVTDQ2PS xmm1,xmm2/mem128     ; 0F 5B /r        [WILLAMETTE,SSE2]
```

CVTDQ2PS converts four packed signed doublewords from the source operand to four packed single-precision FP values in the destination operand.

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

For more details of this instruction, see the Intel Processor manuals.

A.5.37 CVTPD2DQ: Packed Double-Precision FP to Packed Signed INT32 Conversion

```
CVTPD2DQ xmm1,xmm2/mem128     ; F2 0F E6 /r     [WILLAMETTE,SSE2]
```

CVTPD2DQ converts two packed double-precision FP values from the source operand to two packed signed doublewords in the low quadword of the destination operand. The high quadword of the destination is set to all 0s.

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

For more details of this instruction, see the Intel Processor manuals.

A.5.38 `CVTPD2PI`: Packed Double-Precision FP to Packed Signed INT32 Conversion

```
CVTPD2PI mm,xmm/mem128        ; 66 0F 2D /r     [WILLAMETTE,SSE2]
```

`CVTPD2PI` converts two packed double-precision FP values from the source operand to two packed signed doublewords in the destination operand.

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

For more details of this instruction, see the Intel Processor manuals.

A.5.39 `CVTPD2PS`: Packed Double-Precision FP to Packed Single-Precision FP Conversion

```
CVTPD2PS xmm1,xmm2/mem128     ; 66 0F 5A /r     [WILLAMETTE,SSE2]
```

`CVTPD2PS` converts two packed double-precision FP values from the source operand to two packed single-precision FP values in the low quadword of the destination operand. The high quadword of the destination is set to all 0s.

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

For more details of this instruction, see the Intel Processor manuals.

A.5.40 `CVTPI2PD`: Packed Signed INT32 to Packed Double-Precision FP Conversion

```
CVTPI2PD xmm,mm/mem64         ; 66 0F 2A /r     [WILLAMETTE,SSE2]
```

`CVTPI2PD` converts two packed signed doublewords from the source operand to two packed double-precision FP values in the destination operand.

The destination operand is an `XMM` register. The source can be either an `MMX` register or a 64-bit memory location.

For more details of this instruction, see the Intel Processor manuals.

A.5.41 `CVTPI2PS`: Packed Signed INT32 to Packed Single-FP Conversion

```
CVTPI2PS xmm,mm/mem64         ; 0F 2A /r        [KATMAI,SSE]
```

`CVTPI2PS` converts two packed signed doublewords from the source operand to two packed single-precision FP values in the low quadword of the destination operand. The high quadword of the destination remains unchanged.

The destination operand is an `XMM` register. The source can be either an `MMX` register or a 64-bit memory location.

For more details of this instruction, see the Intel Processor manuals.

A.5.42 `CVTPS2DQ`: Packed Single-Precision FP to Packed Signed INT32

Conversion

```
CVTPS2DQ xmm1,xmm2/mem128     ; 66 0F 5B /r     [WILLAMETTE,SSE2]
```

CVTPS2DQ converts four packed single-precision FP values from the source operand to four packed signed doublewords in the destination operand.

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

For more details of this instruction, see the Intel Processor manuals.

A.5.43 CVTPS2PD: Packed Single-Precision FP to Packed Double-Precision FP Conversion

```
CVTPS2PD xmm1,xmm2/mem64      ; 0F 5A /r        [WILLAMETTE,SSE2]
```

CVTPS2PD converts two packed single-precision FP values from the source operand to two packed double-precision FP values in the destination operand.

The destination operand is an XMM register. The source can be either an XMM register or a 64-bit memory location. If the source is a register, the input values are in the low quadword.

For more details of this instruction, see the Intel Processor manuals.

A.5.44 CVTPS2PI: Packed Single-Precision FP to Packed Signed INT32 Conversion

```
CVTPS2PI mm,xmm/mem64         ; 0F 2D /r        [KATMAI,SSE]
```

CVTPS2PI converts two packed single-precision FP values from the source operand to two packed signed doublewords in the destination operand.

The destination operand is an MMX register. The source can be either an XMM register or a 64-bit memory location. If the source is a register, the input values are in the low quadword.

For more details of this instruction, see the Intel Processor manuals.

A.5.45 CVTSD2SI: Scalar Double-Precision FP to Signed INT32 Conversion

```
CVTSD2SI reg32,xmm/mem64      ; F2 0F 2D /r     [WILLAMETTE,SSE2]
```

CVTSD2SI converts a double-precision FP value from the source operand to a signed doubleword in the destination operand.

The destination operand is a general purpose register. The source can be either an XMM register or a 64-bit memory location. If the source is a register, the input value is in the low quadword.

For more details of this instruction, see the Intel Processor manuals.

A.5.46 CVTSD2SS: Scalar Double-Precision FP to Scalar Single-Precision FP Conversion

```
CVTSD2SS xmm1,xmm2/mem64      ; F2 0F 5A /r     [KATMAI,SSE]
```

CVTSD2SS converts a double-precision FP value from the source operand to a single-precision

FP value in the low doubleword of the destination operand. The upper 3 doublewords are left unchanged.

The destination operand is an XMM register. The source can be either an XMM register or a 64-bit memory location. If the source is a register, the input value is in the low quadword.

For more details of this instruction, see the Intel Processor manuals.

A.5.47 CVTSI2SD: Signed INT32 to Scalar Double-Precision FP Conversion

```
CVTSI2SD xmm,r/m32            ; F2 0F 2A /r     [WILLAMETTE,SSE2]
```

CVTSI2SD converts a signed doubleword from the source operand to a double-precision FP value in the low quadword of the destination operand. The high quadword is left unchanged.

The destination operand is an XMM register. The source can be either a general purpose register or a 32-bit memory location.

For more details of this instruction, see the Intel Processor manuals.

A.5.48 CVTSI2SS: Signed INT32 to Scalar Single-Precision FP Conversion

```
CVTSI2SS xmm,r/m32            ; F3 0F 2A /r     [KATMAI,SSE]
```

CVTSI2SS converts a signed doubleword from the source operand to a single-precision FP value in the low doubleword of the destination operand. The upper 3 doublewords are left unchanged.

The destination operand is an XMM register. The source can be either a general purpose register or a 32-bit memory location.

For more details of this instruction, see the Intel Processor manuals.

A.5.49 CVTSS2SD: Scalar Single-Precision FP to Scalar Double-Precision FP Conversion

```
CVTSS2SD xmm1,xmm2/mem32      ; F3 0F 5A /r     [WILLAMETTE,SSE2]
```

CVTSS2SD converts a single-precision FP value from the source operand to a double-precision FP value in the low quadword of the destination operand. The upper quadword is left unchanged.

The destination operand is an XMM register. The source can be either an XMM register or a 32-bit memory location. If the source is a register, the input value is contained in the low doubleword.

For more details of this instruction, see the Intel Processor manuals.

A.5.50 CVTSS2SI: Scalar Single-Precision FP to Signed INT32 Conversion

```
CVTSS2SI reg32,xmm/mem32      ; F3 0F 2D /r     [KATMAI,SSE]
```

CVTSS2SI converts a single-precision FP value from the source operand to a signed doubleword in the destination operand.

The destination operand is a general purpose register. The source can be either an XMM register or a 32-bit memory location. If the source is a register, the input value is in the low doubleword.

For more details of this instruction, see the Intel Processor manuals.

A.5.51 CVTTPD2DQ: Packed Double-Precision FP to Packed Signed INT32 Conversion with Truncation

```
CVTTPD2DQ xmm1,xmm2/mem128    ; 66 0F E6 /r     [WILLAMETTE,SSE2]
```

CVTTPD2DQ converts two packed double-precision FP values in the source operand to two packed single-precision FP values in the destination operand. If the result is inexact, it is truncated (rounded toward zero). The high quadword is set to all 0s.

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

For more details of this instruction, see the Intel Processor manuals.

A.5.52 CVTTPD2PI: Packed Double-Precision FP to Packed Signed INT32 Conversion with Truncation

```
CVTTPD2PI mm,xmm/mem128        ; 66 0F 2C /r     [WILLAMETTE,SSE2]
```

CVTTPD2PI converts two packed double-precision FP values in the source operand to two packed single-precision FP values in the destination operand. If the result is inexact, it is truncated (rounded toward zero).

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

For more details of this instruction, see the Intel Processor manuals.

A.5.53 CVTTPS2DQ: Packed Single-Precision FP to Packed Signed INT32 Conversion with Truncation

```
CVTTPS2DQ xmm1,xmm2/mem128    ; F3 0F 5B /r     [WILLAMETTE,SSE2]
```

CVTTPS2DQ converts four packed single-precision FP values in the source operand to four packed signed doublewords in the destination operand. If the result is inexact, it is truncated (rounded toward zero).

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

For more details of this instruction, see the Intel Processor manuals.

A.5.54 CVTTPS2PI: Packed Single-Precision FP to Packed Signed INT32 Conversion with Truncation

```
CVTTPS2PI mm,xmm/mem64         ; 0F 2C /r        [KATMAI,SSE]
```

CVTTPS2PI converts two packed single-precision FP values in the source operand to two packed signed doublewords in the destination operand. If the result is inexact, it is truncated (rounded toward zero). If the source is a register, the input values are in the low quadword.

The destination operand is an MMX register. The source can be either an XMM register or a 64-bit memory location. If the source is a register, the input value is in the low quadword.

For more details of this instruction, see the Intel Processor manuals.

A.5.55 `CVTTSD2SI`: Scalar Double-Precision FP to Signed INT32 Conversion with Truncation

```
CVTTSD2SI reg32,xmm/mem64      ; F2 0F 2C /r    [WILLAMETTE,SSE2]
```

`CVTTSD2SI` converts a double-precision FP value in the source operand to a signed doubleword in the destination operand. If the result is inexact, it is truncated (rounded toward zero).

The destination operand is a general purpose register. The source can be either an `XMM` register or a 64-bit memory location. If the source is a register, the input value is in the low quadword.

For more details of this instruction, see the Intel Processor manuals.

A.5.56 `CVTTSS2SI`: Scalar Single-Precision FP to Signed INT32 Conversion with Truncation

```
CVTTSD2SI reg32,xmm/mem32      ; F3 0F 2C /r    [KATMAI,SSE]
```

`CVTTSS2SI` converts a single-precision FP value in the source operand to a signed doubleword in the destination operand. If the result is inexact, it is truncated (rounded toward zero).

The destination operand is a general purpose register. The source can be either an `XMM` register or a 32-bit memory location. If the source is a register, the input value is in the low doubleword.

For more details of this instruction, see the Intel Processor manuals.

A.5.57 `DAA`, `DAS`: Decimal Adjustments

```
DAA                           ; 27                     [8086]
DAS                           ; 2F                     [8086]
```

These instructions are used in conjunction with the add and subtract instructions to perform binary-coded decimal arithmetic in *packed* (one BCD digit per nibble) form. For the unpacked equivalents, see section A.5.1.

`DAA` should be used after a one-byte `ADD` instruction whose destination was the `AL` register: by means of examining the value in the `AL` and also the auxiliary carry flag `AF`, it determines whether either digit of the addition has overflowed, and adjusts it (and sets the carry and auxiliary-carry flags) if so. You can add long BCD strings together by doing `ADD`/`DAA` on the low two digits, then doing `ADC`/`DAA` on each subsequent pair of digits.

`DAS` works similarly to `DAA`, but is for use after `SUB` instructions rather than `ADD`.

A.5.58 `DEC`: Decrement Integer

```
DEC reg16                     ; o16 48+r               [8086]
DEC reg32                     ; o32 48+r               [386]
DEC r/m8                      ; FE /1                  [8086]
DEC r/m16                     ; o16 FF /1              [8086]
DEC r/m32                     ; o32 FF /1              [386]
```

`DEC` subtracts 1 from its operand. It does *not* affect the carry flag: to affect the carry flag, use `SUB something,1` (see section A.5.305). `DEC` affects all the other flags according to the result.

This instruction can be used with a `LOCK` prefix to allow atomic execution.

See also `INC` (section A.5.120).

A.5.59 `DIV`: Unsigned Integer Divide

```
DIV r/m8                      ; F6 /6                [8086]
DIV r/m16                     ; o16 F7 /6            [8086]
DIV r/m32                     ; o32 F7 /6            [386]
```

`DIV` performs unsigned integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:

- For `DIV r/m8`, `AX` is divided by the given operand; the quotient is stored in `AL` and the remainder in `AH`.
- For `DIV r/m16`, `DX:AX` is divided by the given operand; the quotient is stored in `AX` and the remainder in `DX`.
- For `DIV r/m32`, `EDX:EAX` is divided by the given operand; the quotient is stored in `EAX` and the remainder in `EDX`.

Signed integer division is performed by the `IDIV` instruction: see section A.5.117.

A.5.60 `DIVPD`: Packed Double-Precision FP Divide

```
DIVPD xmm1,xmm2/mem128        ; 66 0F 5E /r     [WILLAMETTE,SSE2]
```

`DIVPD` divides the two packed double-precision FP values in the destination operand by the two packed double-precision FP values in the source operand, and stores the packed double-precision results in the destination register.

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

```
   dst[0-63]   := dst[0-63]   / src[0-63],
   dst[64-127] := dst[64-127] / src[64-127].
```

A.5.61 `DIVPS`: Packed Single-Precision FP Divide

```
DIVPS xmm1,xmm2/mem128        ; 0F 5E /r        [KATMAI,SSE]
```

`DIVPS` divides the four packed single-precision FP values in the destination operand by the four packed single-precision FP values in the source operand, and stores the packed single-precision results in the destination register.

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

```
   dst[0-31]   := dst[0-31]   / src[0-31],
   dst[32-63]  := dst[32-63]  / src[32-63],
   dst[64-95]  := dst[64-95]  / src[64-95],
   dst[96-127] := dst[96-127] / src[96-127].
```

A.5.62 `DIVSD`: Scalar Double-Precision FP Divide

```
DIVSD xmm1,xmm2/mem64          ; F2 0F 5E /r     [WILLAMETTE,SSE2]
```

`DIVSD` divides the low-order double-precision FP value in the destination operand by the low-order double-precision FP value in the source operand, and stores the double-precision result in the destination register.

The destination is an `XMM` register. The source operand can be either an `XMM` register or a 64-bit memory location.

```
dst[0-63]   := dst[0-63] / src[0-63],
dst[64-127] remains unchanged.
```

A.5.63 `DIVSS`: Scalar Single-Precision FP Divide

```
DIVSS xmm1,xmm2/mem32          ; F3 0F 5E /r     [KATMAI,SSE]
```

`DIVSS` divides the low-order single-precision FP value in the destination operand by the low-order single-precision FP value in the source operand, and stores the single-precision result in the destination register.

The destination is an `XMM` register. The source operand can be either an `XMM` register or a 32-bit memory location.

```
dst[0-31]   := dst[0-31] / src[0-31],
dst[32-127] remains unchanged.
```

A.5.64 `EMMS`: Empty MMX State

```
EMMS                           ; 0F 77                [PENT,MMX]
```

`EMMS` sets the FPU tag word (marking which floating-point registers are available) to all ones, meaning all registers are available for the FPU to use. It should be used after executing `MMX` instructions and before executing any subsequent floating-point operations.

A.5.65 `ENTER`: Create Stack Frame

```
ENTER imm16,imm8               ; C8 iw ib             [186]
```

`ENTER` constructs a stack frame for a high-level language procedure call. The first operand (the `iw` in the opcode definition above refers to the first operand) gives the amount of stack space to allocate for local variables; the second (the `ib` above) gives the nesting level of the procedure (for languages like Pascal, with nested procedures).

The function of `ENTER`, with a nesting level of zero, is equivalent to

```
PUSH EBP              ; or PUSH BP            in 16 bits
MOV EBP, ESP          ; or MOV BP, SP         in 16 bits
LEA ESP, [ESP - op1] ; or LEA SP, [BP - op1] in 16 bits
```

This creates a stack frame with the procedure parameters accessible upwards from `EBP`, and local variables accessible downwards from `EBP`. Note that the flags are not modified by the calculation.

With a nesting level of one, the stack frame created is 4 (or 2) bytes bigger, and the value of the final frame pointer `EBP` is accessible in memory at `[EBP-4]`.

This allows `ENTER`, when called with a nesting level of two, to look at the stack frame described by the *previous* value of `EBP`, find the frame pointer at offset -4 from that, and push it along with its new frame pointer, so that when a level-two procedure is called from within a level-one procedure, `[EBP-4]` holds the frame pointer of the most recent level-one procedure call and `[EBP-8]` holds that of the most recent level-two call. And so on, for nesting levels up to 31. The nesting level is determined by bitwise AND-masking the second operand with 31.

Stack frames created by `ENTER` can be destroyed by the `LEAVE` instruction: see section A.5.136.

A.5.66 F2XM1: Calculate 2**X-1

```
F2XM1                         ; D9 F0                [8086,FPU]
```

`F2XM1` raises 2 to the power of `ST0`, subtracts one, and stores the result back into `ST0`. The initial contents of `ST0` must be a number in the range -1.0 to +1.0.

A.5.67 FABS: Floating-Point Absolute Value

```
FABS                          ; D9 E1                [8086,FPU]
```

`FABS` computes the absolute value of `ST0`,by clearing the sign bit, and stores the result back in `ST0`.

A.5.68 FADD, FADDP: Floating-Point Addition

```
FADD mem32                    ; D8 /0                [8086,FPU]
FADD mem64                    ; DC /0                [8086,FPU]

FADD fpureg                   ; D8 C0+r              [8086,FPU]
FADD ST0,fpureg               ; D8 C0+r              [8086,FPU]

FADD TO fpureg                ; DC C0+r              [8086,FPU]
FADD fpureg,ST0               ; DC C0+r              [8086,FPU]

FADDP fpureg                  ; DE C0+r              [8086,FPU]
FADDP fpureg,ST0              ; DE C0+r              [8086,FPU]
```

- `FADD`, given one operand, adds the operand to `ST0` and stores the result back in `ST0`. If the operand has the `TO` modifier, the result is stored in the register given rather than in `ST0`.
- `FADDP` performs the same function as `FADD TO`, but pops the register stack after storing the result.

The given two-operand forms are synonyms for the one-operand forms.

To add an integer value to `ST0`, use the `FIADD` instruction (section A.5.80).

A.5.69 FBLD, FBSTP: BCD Floating-Point Load and Store

```
FBLD mem80                    ; DF /4                [8086,FPU]
FBSTP mem80                   ; DF /6                [8086,FPU]
```

`FBLD` loads an 80-bit (ten-byte) packed binary-coded decimal number from the given memory address, converts it to a real, and pushes it on the register stack. `FBSTP` stores the value of `ST0`, in packed BCD, at the given address and then pops the register stack.

A.5.70 FCHS: Floating-Point Change Sign

```
FCHS                        ; D9 E0                  [8086,FPU]
```

`FCHS` negates the number in `ST0`, by inverting the sign bit: negative numbers become positive, and vice versa.

A.5.71 FCLEX, FNCLEX: Clear Floating-Point Exceptions

```
FCLEX                       ; 9B DB E2               [8086,FPU]
FNCLEX                      ; DB E2                  [8086,FPU]
```

`FCLEX` clears any floating-point exceptions which may be pending. `FNCLEX` does the same thing but doesn't wait for previous floating-point operations (including the *handling* of pending exceptions) to finish first.

A.5.72 FCMOVcc: Floating-Point Conditional Move

```
FCMOVB fpureg               ; DA C0+r                [P6,FPU]
FCMOVB ST0,fpureg           ; DA C0+r                [P6,FPU]

FCMOVE fpureg               ; DA C8+r                [P6,FPU]
FCMOVE ST0,fpureg           ; DA C8+r                [P6,FPU]

FCMOVBE fpureg              ; DA D0+r                [P6,FPU]
FCMOVBE ST0,fpureg          ; DA D0+r                [P6,FPU]

FCMOVU fpureg               ; DA D8+r                [P6,FPU]
FCMOVU ST0,fpureg           ; DA D8+r                [P6,FPU]

FCMOVNB fpureg              ; DB C0+r                [P6,FPU]
FCMOVNB ST0,fpureg          ; DB C0+r                [P6,FPU]

FCMOVNE fpureg              ; DB C8+r                [P6,FPU]
FCMOVNE ST0,fpureg          ; DB C8+r                [P6,FPU]

FCMOVNBE fpureg             ; DB D0+r                [P6,FPU]
FCMOVNBE ST0,fpureg         ; DB D0+r                [P6,FPU]

FCMOVNU fpureg              ; DB D8+r                [P6,FPU]
FCMOVNU ST0,fpureg          ; DB D8+r                [P6,FPU]
```

The `U` instructions perform conditional move operations: each of them moves the contents of the given register into `ST0` if its condition is satisfied, and does nothing if not.

The conditions are not the same as the standard condition codes used with conditional jump instructions. The conditions `B`, `BE`, `NB`, `NBE`, `E` and `NE` are exactly as normal, but none of the other standard ones are supported. Instead, the condition `U` and its counterpart `NU` are provided; the `U` condition is satisfied if the last two floating-point numbers compared were *unordered*, i.e. they were not equal but neither one could be said to be greater than the other, for example if they were NaNs. (The flag state which signals this is the setting of the parity flag: so the `U` condition is notionally equivalent to `PE`, and `NU` is equivalent to `PO`.)

The `FCMOV` conditions test the main processor's status flags, not the FPU status flags, so using `FCMOV` directly after `FCOM` will not work. Instead, you should either use `FCOMI` which writes directly to the main CPU flags word, or use `FSTSW` to extract the FPU flags.

Although the `FCMOV` instructions are flagged `P6` above, they may not be supported by all Pentium Pro processors; the `CPUID` instruction (section A.5.34) will return a bit which indicates whether conditional moves are supported.

A.5.73 `FCOM`, `FCOMP`, `FCOMPP`, `FCOMI`, `FCOMIP`: Floating-Point Compare

```
FCOM mem32                    ; D8 /2                [8086,FPU]
FCOM mem64                    ; DC /2                [8086,FPU]
FCOM fpureg                   ; D8 D0+r              [8086,FPU]
FCOM ST0,fpureg               ; D8 D0+r              [8086,FPU]

FCOMP mem32                   ; D8 /3                [8086,FPU]
FCOMP mem64                   ; DC /3                [8086,FPU]
FCOMP fpureg                  ; D8 D8+r              [8086,FPU]
FCOMP ST0,fpureg              ; D8 D8+r              [8086,FPU]

FCOMPP                        ; DE D9                [8086,FPU]

FCOMI fpureg                  ; DB F0+r              [P6,FPU]
FCOMI ST0,fpureg              ; DB F0+r              [P6,FPU]

FCOMIP fpureg                 ; DF F0+r              [P6,FPU]
FCOMIP ST0,fpureg             ; DF F0+r              [P6,FPU]
```

`FCOM` compares `ST0` with the given operand, and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that the carry flag is set (for a ‘less-than’ result) if `ST0` is less than the given operand.

`FCOMP` does the same as `FCOM`, but pops the register stack afterwards. `FCOMPP` compares `ST0` with `ST1` and then pops the register stack twice.

`FCOMI` and `FCOMIP` work like the corresponding forms of `FCOM` and `FCOMP`, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The `FCOM` instructions differ from the `FUCOM` instructions (section A.5.108) only in the way they handle quiet NaNs: `FUCOM` will handle them silently and set the condition code flags to an ‘unordered’ result, whereas `FCOM` will generate an exception.

A.5.74 `FCOS`: Cosine

```
FCOS                          ; D9 FF                [386,FPU]
```

`FCOS` computes the cosine of `ST0` (in radians), and stores the result in `ST0`. The absolute value of `ST0` must be less than 2**63.

See also `FSINCOS` (section A.5.100).

A.5.75 FDECSTP: Decrement Floating-Point Stack Pointer

```
FDECSTP                       ; D9 F6                [8086,FPU]
```

FDECSTP decrements the 'top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the contents of ST7 had been pushed on the stack. See also FINCSTP (section A.5.85).

A.5.76 FxDISI, FxENI: Disable and Enable Floating-Point Interrupts

```
FDISI                         ; 9B DB E1             [8086,FPU]
FNDISI                        ; DB E1                [8086,FPU]

FENI                          ; 9B DB E0             [8086,FPU]
FNENI                         ; DB E0                [8086,FPU]
```

FDISI and FENI disable and enable floating-point interrupts. These instructions are only meaningful on original 8087 processors: the 287 and above treat them as no-operation instructions.

FNDISI and FNENI do the same thing as FDISI and FENI respectively, but without waiting for the floating-point processor to finish what it was doing first.

A.5.77 FDIV, FDIVP, FDIVR, FDIVRP: Floating-Point Division

```
FDIV mem32                    ; D8 /6                [8086,FPU]
FDIV mem64                    ; DC /6                [8086,FPU]

FDIV fpureg                   ; D8 F0+r              [8086,FPU]
FDIV ST0,fpureg               ; D8 F0+r              [8086,FPU]

FDIV TO fpureg                ; DC F8+r              [8086,FPU]
FDIV fpureg,ST0               ; DC F8+r              [8086,FPU]

FDIVR mem32                   ; D8 /7                [8086,FPU]
FDIVR mem64                   ; DC /7                [8086,FPU]

FDIVR fpureg                  ; D8 F8+r              [8086,FPU]
FDIVR ST0,fpureg              ; D8 F8+r              [8086,FPU]

FDIVR TO fpureg               ; DC F0+r              [8086,FPU]
FDIVR fpureg,ST0              ; DC F0+r              [8086,FPU]

FDIVP fpureg                  ; DE F8+r              [8086,FPU]
FDIVP fpureg,ST0              ; DE F8+r              [8086,FPU]

FDIVRP fpureg                 ; DE F0+r              [8086,FPU]
FDIVRP fpureg,ST0             ; DE F0+r              [8086,FPU]
```

- FDIV divides ST0 by the given operand and stores the result back in ST0, unless the TO qualifier is given, in which case it divides the given operand by ST0 and stores the result in the operand.
- FDIVR does the same thing, but does the division the other way up: so if TO is not given, it divides the given operand by ST0 and stores the result in ST0, whereas if TO is given it

divides `ST0` by its operand and stores the result in the operand.

- `FDIVP` operates like `FDIV TO`, but pops the register stack once it has finished.
- `FDIVRP` operates like `FDIVR TO`, but pops the register stack once it has finished.

For FP/Integer divisions, see `FIDIV` (section A.5.82).

A.5.78 FEMMS: Faster Enter/Exit of the MMX or floating-point state

```
FEMMS                          ; 0F 0E           [PENT,3DNOW]
```

`FEMMS` can be used in place of the `EMMS` instruction on processors which support the 3DNow! instruction set. Following execution of `FEMMS`, the state of the `MMX/FP` registers is undefined, and this allows a faster context switch between `FP` and `MMX` instructions. The `FEMMS` instruction can also be used *before* executing `MMX` instructions.

A.5.79 FFREE: Flag Floating-Point Register as Unused

```
FFREE fpureg                   ; DD C0+r              [8086,FPU]
FFREEP fpureg                  ; DF C0+r              [286,FPU,UNDOC]
```

`FFREE` marks the given register as being empty.

`FFREEP` marks the given register as being empty, and then pops the register stack.

A.5.80 FIADD: Floating-Point/Integer Addition

```
FIADD mem16                    ; DE /0                [8086,FPU]
FIADD mem32                    ; DA /0                [8086,FPU]
```

`FIADD` adds the 16-bit or 32-bit integer stored in the given memory location to `ST0`, storing the result in `ST0`.

A.5.81 FICOM, FICOMP: Floating-Point/Integer Compare

```
FICOM mem16                    ; DE /2                [8086,FPU]
FICOM mem32                    ; DA /2                [8086,FPU]

FICOMP mem16                   ; DE /3                [8086,FPU]
FICOMP mem32                   ; DA /3                [8086,FPU]
```

`FICOM` compares `ST0` with the 16-bit or 32-bit integer stored in the given memory location, and sets the FPU flags accordingly. `FICOMP` does the same, but pops the register stack afterwards.

A.5.82 FIDIV, FIDIVR: Floating-Point/Integer Division

```
FIDIV mem16                    ; DE /6                [8086,FPU]
FIDIV mem32                    ; DA /6                [8086,FPU]

FIDIVR mem16                   ; DE /7                [8086,FPU]
FIDIVR mem32                   ; DA /7                [8086,FPU]
```

`FIDIV` divides `ST0` by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in `ST0`. `FIDIVR` does the division the other way up: it divides the integer by `ST0`, but still stores the result in `ST0`.

A.5.83 `FILD`, `FIST`, `FISTP`: Floating-Point/Integer Conversion

```
FILD mem16                    ; DF /0                [8086,FPU]
FILD mem32                    ; DB /0                [8086,FPU]
FILD mem64                    ; DF /5                [8086,FPU]

FIST mem16                    ; DF /2                [8086,FPU]
FIST mem32                    ; DB /2                [8086,FPU]

FISTP mem16                   ; DF /3                [8086,FPU]
FISTP mem32                   ; DB /3                [8086,FPU]
FISTP mem64                   ; DF /7                [8086,FPU]
```

`FILD` loads an integer out of a memory location, converts it to a real, and pushes it on the FPU register stack. `FIST` converts `ST0` to an integer and stores that in memory; `FISTP` does the same as `FIST`, but pops the register stack afterwards.

A.5.84 `FIMUL`: Floating-Point/Integer Multiplication

```
FIMUL mem16                   ; DE /1                [8086,FPU]
FIMUL mem32                   ; DA /1                [8086,FPU]
```

`FIMUL` multiplies `ST0` by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in `ST0`.

A.5.85 `FINCSTP`: Increment Floating-Point Stack Pointer

```
FINCSTP                       ; D9 F7                [8086,FPU]
```

`FINCSTP` increments the 'top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the register stack had been popped; however, unlike the popping of the stack performed by many FPU instructions, it does not flag the new `ST7` (previously `ST0`) as empty. See also `FDECSTP` (section A.5.75).

A.5.86 `FINIT`, `FNINIT`: initialize Floating-Point Unit

```
FINIT                         ; 9B DB E3             [8086,FPU]
FNINIT                        ; DB E3                [8086,FPU]
```

`FINIT` initializes the FPU to its default state. It flags all registers as empty, without actually change their values, clears the top of stack pointer. `FNINIT` does the same, without first waiting for pending exceptions to clear.

A.5.87 `FISUB`: Floating-Point/Integer Subtraction

```
FISUB mem16                   ; DE /4                [8086,FPU]
FISUB mem32                   ; DA /4                [8086,FPU]

FISUBR mem16                  ; DE /5                [8086,FPU]
FISUBR mem32                  ; DA /5                [8086,FPU]
```

`FISUB` subtracts the 16-bit or 32-bit integer stored in the given memory location from `ST0`, and stores the result in `ST0`. `FISUBR` does the subtraction the other way round, i.e. it subtracts `ST0` from the given integer, but still stores the result in `ST0`.

A.5.88 FLD: Floating-Point Load

```
FLD mem32                     ; D9 /0               [8086,FPU]
FLD mem64                     ; DD /0               [8086,FPU]
FLD mem80                     ; DB /5               [8086,FPU]
FLD fpureg                    ; D9 C0+r             [8086,FPU]
```

FLD loads a floating-point value out of the given register or memory location, and pushes it on the FPU register stack.

A.5.89 FLDxx: Floating-Point Load Constants

```
FLD1                          ; D9 E8               [8086,FPU]
FLDL2E                        ; D9 EA               [8086,FPU]
FLDL2T                        ; D9 E9               [8086,FPU]
FLDLG2                        ; D9 EC               [8086,FPU]
FLDLN2                        ; D9 ED               [8086,FPU]
FLDPI                         ; D9 EB               [8086,FPU]
FLDZ                          ; D9 EE               [8086,FPU]
```

These instructions push specific standard constants on the FPU register stack.

```
 Instruction    Constant pushed

 FLD1           1
 FLDL2E         base-2 logarithm of e
 FLDL2T         base-2 log of 10
 FLDLG2         base-10 log of 2
 FLDLN2         base-e log of 2
 FLDPI          pi
 FLDZ           zero
```

A.5.90 FLDCW: Load Floating-Point Control Word

```
FLDCW mem16                   ; D9 /5               [8086,FPU]
```

FLDCW loads a 16-bit value out of memory and stores it into the FPU control word (governing things like the rounding mode, the precision, and the exception masks). See also FSTCW (section A.5.103). If exceptions are enabled and you don't want to generate one, use FCLEX or FNCLEX (section A.5.71) before loading the new control word.

A.5.91 FLDENV: Load Floating-Point Environment

```
FLDENV mem                    ; D9 /4               [8086,FPU]
```

FLDENV loads the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) from memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also FSTENV (section A.5.104).

A.5.92 FMUL, FMULP: Floating-Point Multiply

```
FMUL mem32                    ; D8 /1               [8086,FPU]
FMUL mem64                    ; DC /1               [8086,FPU]
```

```
FMUL fpureg                   ; D8 C8+r                [8086,FPU]
FMUL ST0,fpureg               ; D8 C8+r                [8086,FPU]

FMUL TO fpureg                ; DC C8+r                [8086,FPU]
FMUL fpureg,ST0               ; DC C8+r                [8086,FPU]

FMULP fpureg                  ; DE C8+r                [8086,FPU]
FMULP fpureg,ST0              ; DE C8+r                [8086,FPU]
```

`FMUL` multiplies `ST0` by the given operand, and stores the result in `ST0`, unless the `TO` qualifier is used in which case it stores the result in the operand. `FMULP` performs the same operation as `FMUL TO`, and then pops the register stack.

A.5.93 FNOP: Floating-Point No Operation

```
FNOP                          ; D9 D0                  [8086,FPU]
```

`FNOP` does nothing.

A.5.94 FPATAN, FPTAN: Arctangent and Tangent

```
FPATAN                        ; D9 F3                  [8086,FPU]
FPTAN                         ; D9 F2                  [8086,FPU]
```

`FPATAN` computes the arctangent, in radians, of the result of dividing `ST1` by `ST0`, stores the result in `ST1`, and pops the register stack. It works like the C `atan2` function, in that changing the sign of both `ST0` and `ST1` changes the output value by pi (so it performs true rectangular-to-polar coordinate conversion, with `ST1` being the Y coordinate and `ST0` being the X coordinate, not merely an arctangent).

`FPTAN` computes the tangent of the value in `ST0` (in radians), and stores the result back into `ST0`.

The absolute value of `ST0` must be less than 2**63.

A.5.95 FPREM, FPREM1: Floating-Point Partial Remainder

```
FPREM                         ; D9 F8                  [8086,FPU]
FPREM1                        ; D9 F5                  [386,FPU]
```

These instructions both produce the remainder obtained by dividing `ST0` by `ST1`. This is calculated, notionally, by dividing `ST0` by `ST1`, rounding the result to an integer, multiplying by `ST1` again, and computing the value which would need to be added back on to the result to get back to the original value in `ST0`.

The two instructions differ in the way the notional round-to-integer operation is performed. `FPREM` does it by rounding towards zero, so that the remainder it returns always has the same sign as the original value in `ST0`; `FPREM1` does it by rounding to the nearest integer, so that the remainder always has at most half the magnitude of `ST1`.

Both instructions calculate *partial* remainders, meaning that they may not manage to provide the final result, but might leave intermediate results in `ST0` instead. If this happens, they will set the C2 flag in the FPU status word; therefore, to calculate a remainder, you should repeatedly execute `FPREM` or `FPREM1` until C2 becomes clear.

A.5.96 FRNDINT: Floating-Point Round to Integer

```
FRNDINT                       ; D9 FC                  [8086,FPU]
```

FRNDINT rounds the contents of ST0 to an integer, according to the current rounding mode set in the FPU control word, and stores the result back in ST0.

A.5.97 FSAVE, FRSTOR: Save/Restore Floating-Point State

```
FSAVE mem                     ; 9B DD /6               [8086,FPU]
FNSAVE mem                    ; DD /6                  [8086,FPU]

FRSTOR mem                    ; DD /4                  [8086,FPU]
```

FSAVE saves the entire floating-point unit state, including all the information saved by FSTENV (section A.5.104) plus the contents of all the registers, to a 94 or 108 byte area of memory (depending on the CPU mode). FRSTOR restores the floating-point state from the same area of memory.

FNSAVE does the same as FSAVE, without first waiting for pending floating-point exceptions to clear.

A.5.98 FSCALE: Scale Floating-Point Value by Power of Two

```
FSCALE                        ; D9 FD                  [8086,FPU]
```

FSCALE scales a number by a power of two: it rounds ST1 towards zero to obtain an integer, then multiplies ST0 by two to the power of that integer, and stores the result in ST0.

A.5.99 FSETPM: Set Protected Mode

```
FSETPM                        ; DB E4                  [286,FPU]
```

This instruction initializes protected mode on the 287 floating-point coprocessor. It is only meaningful on that processor: the 387 and above treat the instruction as a no-operation.

A.5.100 FSIN, FSINCOS: Sine and Cosine

```
FSIN                          ; D9 FE                  [386,FPU]
FSINCOS                       ; D9 FB                  [386,FPU]
```

FSIN calculates the sine of ST0 (in radians) and stores the result in ST0. FSINCOS does the same, but then pushes the cosine of the same value on the register stack, so that the sine ends up in ST1 and the cosine in ST0. FSINCOS is faster than executing FSIN and FCOS (see section A.5.74) in succession.

The absolute value of ST0 must be less than 2**63.

A.5.101 FSQRT: Floating-Point Square Root

```
FSQRT                         ; D9 FA                  [8086,FPU]
```

FSQRT calculates the square root of ST0 and stores the result in ST0.

A.5.102 FST, FSTP: Floating-Point Store

```
FST mem32                     ; D9 /2                  [8086,FPU]
FST mem64                     ; DD /2                  [8086,FPU]
FST fpureg                    ; DD D0+r                [8086,FPU]

FSTP mem32                    ; D9 /3                  [8086,FPU]
FSTP mem64                    ; DD /3                  [8086,FPU]
FSTP mem80                    ; DB /7                  [8086,FPU]
FSTP fpureg                   ; DD D8+r                [8086,FPU]
```

`FST` stores the value in `ST0` into the given memory location or other FPU register. `FSTP` does the same, but then pops the register stack.

A.5.103 `FSTCW`: Store Floating-Point Control Word

```
FSTCW mem16                   ; 9B D9 /7               [8086,FPU]
FNSTCW mem16                  ; D9 /7                  [8086,FPU]
```

`FSTCW` stores the `FPU` control word (governing things like the rounding mode, the precision, and the exception masks) into a 2-byte memory area. See also `FLDCW` (section A.5.90).

`FNSTCW` does the same thing as `FSTCW`, without first waiting for pending floating-point exceptions to clear.

A.5.104 `FSTENV`: Store Floating-Point Environment

```
FSTENV mem                    ; 9B D9 /6               [8086,FPU]
FNSTENV mem                   ; D9 /6                  [8086,FPU]
```

`FSTENV` stores the `FPU` operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) into memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also `FLDENV` (section A.5.91).

`FNSTENV` does the same thing as `FSTENV`, without first waiting for pending floating-point exceptions to clear.

A.5.105 `FSTSW`: Store Floating-Point Status Word

```
FSTSW mem16                   ; 9B DD /7               [8086,FPU]
FSTSW AX                      ; 9B DF E0               [286,FPU]

FNSTSW mem16                  ; DD /7                  [8086,FPU]
FNSTSW AX                     ; DF E0                  [286,FPU]
```

`FSTSW` stores the `FPU` status word into `AX` or into a 2-byte memory area.

`FNSTSW` does the same thing as `FSTSW`, without first waiting for pending floating-point exceptions to clear.

A.5.106 `FSUB`, `FSUBP`, `FSUBR`, `FSUBRP`: Floating-Point Subtract

```
FSUB mem32                    ; D8 /4                  [8086,FPU]
FSUB mem64                    ; DC /4                  [8086,FPU]

FSUB fpureg                   ; D8 E0+r                [8086,FPU]
FSUB ST0,fpureg               ; D8 E0+r                [8086,FPU]
```

```
FSUB TO fpureg                ; DC E8+r                [8086,FPU]
FSUB fpureg,ST0               ; DC E8+r                [8086,FPU]

FSUBR mem32                   ; D8 /5                  [8086,FPU]
FSUBR mem64                   ; DC /5                  [8086,FPU]

FSUBR fpureg                  ; D8 E8+r                [8086,FPU]
FSUBR ST0,fpureg              ; D8 E8+r                [8086,FPU]

FSUBR TO fpureg               ; DC E0+r                [8086,FPU]
FSUBR fpureg,ST0              ; DC E0+r                [8086,FPU]

FSUBP fpureg                  ; DE E8+r                [8086,FPU]
FSUBP fpureg,ST0              ; DE E8+r                [8086,FPU]

FSUBRP fpureg                 ; DE E0+r                [8086,FPU]
FSUBRP fpureg,ST0             ; DE E0+r                [8086,FPU]
```

- `FSUB` subtracts the given operand from `ST0` and stores the result back in `ST0`, unless the `TO` qualifier is given, in which case it subtracts `ST0` from the given operand and stores the result in the operand.
- `FSUBR` does the same thing, but does the subtraction the other way up: so if `TO` is not given, it subtracts `ST0` from the given operand and stores the result in `ST0`, whereas if `TO` is given it subtracts its operand from `ST0` and stores the result in the operand.
- `FSUBP` operates like `FSUB TO`, but pops the register stack once it has finished.
- `FSUBRP` operates like `FSUBR TO`, but pops the register stack once it has finished.

A.5.107 FTST: Test ST0 Against Zero

```
FTST                          ; D9 E4                  [8086,FPU]
```

`FTST` compares `ST0` with zero and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that a 'less-than' result is generated if `ST0` is negative.

A.5.108 FUCOMxx: Floating-Point Unordered Compare

```
FUCOM fpureg                  ; DD E0+r                [386,FPU]
FUCOM ST0,fpureg              ; DD E0+r                [386,FPU]

FUCOMP fpureg                 ; DD E8+r                [386,FPU]
FUCOMP ST0,fpureg             ; DD E8+r                [386,FPU]

FUCOMPP                       ; DA E9                  [386,FPU]

FUCOMI fpureg                 ; DB E8+r                [P6,FPU]
FUCOMI ST0,fpureg             ; DB E8+r                [P6,FPU]

FUCOMIP fpureg                ; DF E8+r                [P6,FPU]
FUCOMIP ST0,fpureg            ; DF E8+r                [P6,FPU]
```

- `FUCOM` compares `ST0` with the given operand, and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that the carry flag is set (for a 'less-than' result) if ST0 is less than the given operand.

- FUCOMP does the same as FUCOM, but pops the register stack afterwards. FUCOMPP compares ST0 with ST1 and then pops the register stack twice.
- FUCOMI and FUCOMIP work like the corresponding forms of FUCOM and FUCOMP, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The FUCOM instructions differ from the FCOM instructions (section A.5.73) only in the way they handle quiet NaNs: FUCOM will handle them silently and set the condition code flags to an 'unordered' result, whereas FCOM will generate an exception.

A.5.109 FXAM: Examine Class of Value in ST0

```
FXAM                          ; D9 E5                [8086,FPU]
```

FXAM sets the FPU flags C3, C2 and C0 depending on the type of value stored in ST0:

```
Register contents      Flags

Unsupported format     000
NaN                    001
Finite number          010
Infinity               011
Zero                   100
Empty register         101
Denormal               110
```

Additionally, the C1 flag is set to the sign of the number.

A.5.110 FXCH: Floating-Point Exchange

```
FXCH                          ; D9 C9                [8086,FPU]
FXCH fpureg                   ; D9 C8+r              [8086,FPU]
FXCH fpureg,ST0               ; D9 C8+r              [8086,FPU]
FXCH ST0,fpureg               ; D9 C8+r              [8086,FPU]
```

FXCH exchanges ST0 with a given FPU register. The no-operand form exchanges ST0 with ST1.

A.5.111 FXRSTOR: Restore FP, MMX and SSE State

```
FXRSTOR memory                ; 0F AE /1               [P6,SSE,FPU]
```

The FXRSTOR instruction reloads the FPU, MMX and SSE state (environment and registers), from the 512 byte memory area defined by the source operand. This data should have been written by a previous FXSAVE.

A.5.112 FXSAVE: Store FP, MMX and SSE State

```
FXSAVE memory                 ; 0F AE /0         [P6,SSE,FPU]
```

The FXSAVE instruction writes the current FPU, MMX and SSE technology states (environment and registers), to the 512 byte memory area defined by the destination operand. It does this without checking for pending unmasked floating-point exceptions (similar to the operation of FNSAVE).

Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FPU, MMX and SSE state in the processor after the state has been saved. This instruction has been optimized to maximize floating-point save performance.

A.5.113 FXTRACT: Extract Exponent and Significand

```
FXTRACT                       ; D9 F4                 [8086,FPU]
```

FXTRACT separates the number in ST0 into its exponent and significand (mantissa), stores the exponent back into ST0, and then pushes the significand on the register stack (so that the significand ends up in ST0, and the exponent in ST1).

A.5.114 FYL2X, FYL2XP1: Compute Y times Log2(X) or Log2(X+1)

```
FYL2X                         ; D9 F1                 [8086,FPU]
FYL2XP1                       ; D9 F9                 [8086,FPU]
```

FYL2X multiplies ST1 by the base-2 logarithm of ST0, stores the result in ST1, and pops the register stack (so that the result ends up in ST0). ST0 must be non-zero and positive.

FYL2XP1 works the same way, but replacing the base-2 log of ST0 with that of ST0 plus one. This time, ST0 must have magnitude no greater than 1 minus half the square root of two.

A.5.115 HLT: Halt Processor

```
HLT                           ; F4                    [8086,PRIV]
```

HLT puts the processor into a halted state, where it will perform no more operations until restarted by an interrupt or a reset.

On the 286 and later processors, this is a privileged instruction.

This instruction, when supported, generally causes the CPU to idle, either by sleeping a certain amount of time, releasing a time slice in a multitasker, or actually halting until the next hardware interrupt occurs. Thus, executing HLT within an input loop after not receiving any new inputs allows to idle the system.

DPMI environments may fault when trying to execute HLT, if the host does not support this usage. Calls such as interrupt 2Fh with AX=1680h must be used instead then.

A.5.116 IBTS: Insert Bit String

```
IBTS r/m16,reg16              ; o16 0F A7 /r          [386,UNDOC]
IBTS r/m32,reg32              ; o32 0F A7 /r          [386,UNDOC]
```

The implied operation of this instruction is:

```
IBTS r/m16,AX,CL,reg16
IBTS r/m32,EAX,CL,reg32
```

Writes a bit string from the source operand to the destination. CL indicates the number of bits to be copied, from the low bits of the source. (E)AX indicates the low order bit offset in the destination that is written to. For example, if CL is set to 4 and AX (for 16-bit code) is set to 5, bits 0-3 of `src` will be copied to bits 5-8 of `dst`. This instruction is very poorly documented, and I have been unable to find any official source of documentation on it.

`IBTS` is supported only on the early Intel 386s, and conflicts with the opcodes for `CMPXCHG486` (on early Intel 486s). NASM supports it only for completeness. Its counterpart is `XBTS` (see section A.5.332).

A.5.117 `IDIV`: Signed Integer Divide

```
IDIV r/m8                     ; F6 /7                  [8086]
IDIV r/m16                    ; o16 F7 /7              [8086]
IDIV r/m32                    ; o32 F7 /7              [386]
```

`IDIV` performs signed integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:

- For `IDIV r/m8`, `AX` is divided by the given operand; the quotient is stored in `AL` and the remainder in `AH`.
- For `IDIV r/m16`, `DX:AX` is divided by the given operand; the quotient is stored in `AX` and the remainder in `DX`.
- For `IDIV r/m32`, `EDX:EAX` is divided by the given operand; the quotient is stored in `EAX` and the remainder in `EDX`.

Unsigned integer division is performed by the `DIV` instruction: see section A.5.59.

A.5.118 `IMUL`: Signed Integer Multiply

```
IMUL r/m8                     ; F6 /5                  [8086]
IMUL r/m16                    ; o16 F7 /5              [8086]
IMUL r/m32                    ; o32 F7 /5              [386]

IMUL reg16,r/m16              ; o16 0F AF /r           [386]
IMUL reg32,r/m32              ; o32 0F AF /r           [386]

IMUL reg16,imm8               ; o16 6B /r ib           [186]
IMUL reg16,imm16              ; o16 69 /r iw           [186]
IMUL reg32,imm8               ; o32 6B /r ib           [386]
IMUL reg32,imm32              ; o32 69 /r id           [386]

IMUL reg16,r/m16,imm8         ; o16 6B /r ib           [186]
IMUL reg16,r/m16,imm16        ; o16 69 /r iw           [186]
IMUL reg32,r/m32,imm8         ; o32 6B /r ib           [386]
IMUL reg32,r/m32,imm32        ; o32 69 /r id           [386]
```

`IMUL` performs signed integer multiplication. For the single-operand form, the other operand and destination are implicit, in the following way:

- For `IMUL r/m8`, `AL` is multiplied by the given operand; the product is stored in `AX`.
- For `IMUL r/m16`, `AX` is multiplied by the given operand; the product is stored in `DX:AX`.
- For `IMUL r/m32`, `EAX` is multiplied by the given operand; the product is stored in `EDX:EAX`.

The two-operand form multiplies its two operands and stores the result in the destination (first) operand. The three-operand form multiplies its last two operands and stores the result in the first operand.

The two-operand form with an immediate second operand is in fact a shorthand for the three-operand form, as can be seen by examining the opcode descriptions: in the two-operand form, the code `/r` takes both its register and `r/m` parts from the same operand (the first one).

In the forms with an 8-bit immediate operand and another longer source operand, the immediate operand is considered to be signed, and is sign-extended to the length of the other source operand. The `BYTE` qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the `STRICT WORD` or `STRICT DWORD` qualifier.

Unsigned integer multiplication is performed by the `MUL` instruction: see section A.5.184.

A.5.119 `IN`: Input from I/O Port

```
IN AL,imm8                      ; E4 ib                  [8086]
IN AX,imm8                      ; o16 E5 ib              [8086]
IN EAX,imm8                     ; o32 E5 ib              [386]
IN AL,DX                        ; EC                     [8086]
IN AX,DX                        ; o16 ED                 [8086]
IN EAX,DX                       ; o32 ED                 [386]
```

`IN` reads a byte, word or doubleword from the specified I/O port, and stores it in the given destination register. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in `DX`. See also `OUT` (section A.5.194).

A.5.120 `INC`: Increment Integer

```
INC reg16                       ; o16 40+r               [8086]
INC reg32                       ; o32 40+r               [386]
INC r/m8                        ; FE /0                  [8086]
INC r/m16                       ; o16 FF /0              [8086]
INC r/m32                       ; o32 FF /0              [386]
```

`INC` adds 1 to its operand. It does *not* affect the carry flag: to affect the carry flag, use `ADD something,1` (see section A.5.3). `INC` affects all the other flags according to the result.

This instruction can be used with a `LOCK` prefix to allow atomic execution.

See also `DEC` (section A.5.58).

A.5.121 `INSB`, `INSW`, `INSD`: Input String from I/O Port

```
INSB                            ; 6C                     [186]
INSW                            ; o16 6D                 [186]
INSD                            ; o32 6D                 [386]
```

`INSB` inputs a byte from the I/O port specified in `DX` and stores it at `[ES:DI]` or `[ES:EDI]`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` or `EDI`. The flags are not modified.

The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

Segment override prefixes have no effect for this instruction: the use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.

`INSW` and `INSD` work in the same way, but they input a word or a doubleword instead of a byte, and increment or decrement the addressing register by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` - again, the address size chooses which) times. The `REP` alias `REPE`, as well as the differently-encoded `REPNE`, are both allowed as well. They behave in the same way as `REP`.

See also `OUTSB`, `OUTSW` and `OUTSD` (section A.5.195).

Note that `INSB` is equivalent to `IN AL,DX` followed by `STOSB`, except it doesn't write the accumulator register (AL).

A.5.121.1 Pseudo-code examples

`a16 INSB` and with Direction Flag clear (UP) is equal to

```
IN BYTE [ES:DI], DX
LEA DI, [DI + 1]
```

`a16 REP INSW` and with Direction Flag clear (UP) is equal to

```
JCXZ @FF
@@:
IN WORD [ES:DI], DX
LEA DI, [DI + 2]
a16 LOOP @B
@@:
```

`a32 INSD` with Direction Flag set (DN) is equal to

```
IN DWORD [ES:EDI], DX
LEA EDI, [EDI - 4]
```

A.5.122 `INT`: Software Interrupt

```
INT imm8                        ; CD ib                 [8086]
```

`INT` causes a software interrupt through a specified vector number from 0 to 255.

The code generated by the `INT` instruction is always two bytes long: although there are short forms for some `INT` instructions, NASM does not generate them when it sees the `INT` mnemonic. In order to generate single-byte breakpoint instructions, use the `INT3` or `INT1` instructions (see section A.5.123) instead.

A.5.123 `INT3`, `INT1`, `ICEBP`, `INT01`: Breakpoints

```
INT1                            ; F1                    [P6]
ICEBP                           ; F1                    [P6]
INT01                           ; F1                    [P6]
```

```
INT3                              ; CC                     [8086]
INT03                             ; CC                     [8086]
```

`INT1` and `INT3` are short one-byte forms of the instructions `INT 1` and `INT 3` (see section A.5.122). They perform a similar function to their longer counterparts, but take up less code space. They are used as breakpoints by debuggers.

- `INT1`, and its alternative synonyms `INT01` and `ICEBP`, is an instruction used by in-circuit emulators (ICEs). It is present, though not documented, on some processors down to the 286, but is only documented for the Pentium Pro. `INT3` is the instruction normally used as a breakpoint by debuggers.

- `INT3`, and its synonym `INT03`, is not precisely equivalent to `INT 3`: the short form, since it is designed to be used as a breakpoint, bypasses the normal `IOPL` checks in virtual-8086 mode, and also does not go through interrupt redirection.

A.5.124 INTO: Interrupt if Overflow

```
INTO                              ; CE                     [8086]
```

`INTO` performs an `INT 4` software interrupt (see section A.5.122) if and only if the overflow flag is set.

A.5.125 INVD: Invalidate Internal Caches

```
INVD                              ; 0F 08                  [486]
```

`INVD` invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It does not write the contents of the caches back to memory first: any modified data held in the caches will be lost. To write the data back first, use `WBINVD` (section A.5.328).

A.5.126 INVLPG: Invalidate TLB Entry

```
INVLPG mem                        ; 0F 01 /7               [486]
```

`INVLPG` invalidates the translation lookahead buffer (TLB) entry associated with the supplied memory address.

A.5.127 IRET, IRETW, IRETD: Return from Interrupt

```
IRET                              ; CF                     [8086]
IRETW                             ; o16 CF                 [8086]
IRETD                             ; o32 CF                 [386]
```

`IRET` returns from an interrupt (hardware or software) by means of popping `IP` (or `EIP`), `CS` and the flags off the stack and then continuing execution from the new `CS:IP`.

`IRETW` pops `IP`, `CS` and the flags as 2 bytes each, taking 6 bytes off the stack in total. `IRETD` pops `EIP` as 4 bytes, pops a further 4 bytes of which the top two are discarded and the bottom two go into `CS`, and pops the flags as 4 bytes as well, taking 12 bytes off the stack.

`IRET` is a shorthand for either `IRETW` or `IRETD`, depending on the default `BITS` setting at the time.

A.5.128 `Jcc`: Conditional Branch

```
Jcc imm                       ; 70+cc rb              [8086]
Jcc NEAR imm                  ; 0F 80+cc rw/rd        [386]
Jcc NEAR imm                  ; 70+(cc^1) 03 E9 rw    [8086]
```

The conditional jump instructions execute a near (same segment) jump if and only if their conditions are satisfied. For example, `JNZ` jumps only if the zero flag is not set.

The ordinary form of the instructions has only a 128-byte range. The single-instruction `NEAR` form is a 386 extension to the instruction set, and can span the full size of a segment. When `CPU` is set to 386, NASM will automatically choose the single-instruction `NEAR` form when the jump exceeds the `SHORT` range.

When `CPU` is set to below 386 (any of 286, 186, 8086), recent versions of NASM will generate two instructions to work around the range limitation. The first instruction will be a short conditional jump of the opposite condition code as the one desired. This first jump will (if taken) jump to behind the second instruction, which is an unconditional near jump. (Only the 16-bit variant is shown for this, because 32-bit assembly means that the single-instruction form is available.)

You can override the choice of jump instruction using an explicit `SHORT` keyword, which will cause an error if the jump target is out of range. Also, an explicit `STRICT NEAR` qualifier makes NASM always use the single-instruction near jump, even if not needed. (If `CPU` is set to below 386, this will result in an error.) There is no way to force the work around pair of instructions; if you want these unconditionally, you have to code them manually.

For details of the condition codes, see section A.2.2.

A.5.129 `JCXZ`, `JECXZ`: Jump if CX/ECX Zero

```
JCXZ imm                      ; a16 E3 rb             [8086]
JECXZ imm                     ; a32 E3 rb             [386]
```

`JCXZ` performs a short jump (with maximum range 128 bytes) if and only if the contents of the `CX` register is 0. `JECXZ` does the same thing, but with `ECX`.

A.5.130 `JMP`: Jump

```
JMP imm                       ; E9 rw/rd              [8086]
JMP SHORT imm                 ; EB rb                 [8086]
JMP imm:imm16                 ; o16 EA iw iw          [8086]
JMP imm:imm32                 ; o32 EA id iw          [386]
JMP FAR mem                   ; o16 FF /5             [8086]
JMP FAR mem32                 ; o32 FF /5             [386]
JMP r/m16                     ; o16 FF /4             [8086]
JMP r/m32                     ; o32 FF /4             [386]
```

`JMP` jumps to a given address. The address may be specified as an absolute segment and offset, or as a relative jump within the current segment.

`JMP SHORT imm` has a maximum range of 128 bytes, since the displacement is specified as only 8 bits, but takes up less code space. Recent versions of NASM automatically generate a `JMP SHORT` for you when the target is in range and known during assembling (ie, before linking). Specifying the `SHORT` keyword explicitly will cause an error if the jump target is out of range. Specifying a `STRICT NEAR` qualifier forces NASM to assemble a near jump, even if the target is in range of a short jump.

You can choose between the two immediate far jump forms (`JMP imm:imm`) by the use of the `WORD` and `DWORD` keywords: `JMP WORD 0x1234:0x5678` or `JMP DWORD 0x1234:0x56789abc`.

The `JMP FAR mem` forms execute a far jump by loading the destination address out of memory. The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using `JMP WORD FAR mem` or `JMP DWORD FAR mem`.

The `JMP r/m` forms execute a near jump (within the same segment), loading the destination address out of memory or out of a register. The keyword `NEAR` may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using `JMP WORD mem` or `JMP DWORD mem`.

As a convenience, NASM does not require you to jump to a far symbol by coding the cumbersome `JMP SEG routine:routine`, but instead allows the easier synonym `JMP FAR routine`.

A.5.131 `LAHF`: Load AH from Flags

```
LAHF                          ; 9F                   [8086]
```

`LAHF` sets the `AH` register according to the contents of the low byte of the flags word.

The operation of `LAHF` is:

```
 AH <-- SF:ZF:0:AF:0:PF:1:CF
```

See also `SAHF` (section A.5.282).

A.5.132 `LAR`: Load Access Rights

```
LAR reg16,r/m16               ; o16 0F 02 /r         [286,PRIV]
LAR reg32,r/m32               ; o32 0F 02 /r         [386,PRIV]
```

`LAR` takes the segment selector specified by its source (second) operand, finds the corresponding segment descriptor in the GDT or LDT, and loads the access-rights byte of the descriptor into its destination (first) operand.

A.5.133 `LDMXCSR`: Load Streaming SIMD Extension Control/Status

```
LDMXCSR mem32                 ; 0F AE /2        [KATMAI,SSE]
```

`LDMXCSR` loads 32-bits of data from the specified memory location into the `MXCSR` control/status register. `MXCSR` is used to enable masked/unmasked exception handling, to set rounding modes, to set flush-to-zero mode, and to view exception status flags.

For details of the `MXCSR` register, see the Intel processor docs.

See also `STMXCSR` (section A.5.302

A.5.134 LDS, LES, LFS, LGS, LSS: Load Far Pointer

```
LDS reg16,mem                 ; o16 C5 /r            [8086]
LDS reg32,mem                 ; o32 C5 /r            [386]

LES reg16,mem                 ; o16 C4 /r            [8086]
LES reg32,mem                 ; o32 C4 /r            [386]

LFS reg16,mem                 ; o16 0F B4 /r         [386]
LFS reg32,mem                 ; o32 0F B4 /r         [386]

LGS reg16,mem                 ; o16 0F B5 /r         [386]
LGS reg32,mem                 ; o32 0F B5 /r         [386]

LSS reg16,mem                 ; o16 0F B2 /r         [386]
LSS reg32,mem                 ; o32 0F B2 /r         [386]
```

These instructions load an entire far pointer (16 or 32 bits of offset, plus 16 bits of segment) out of memory in one go. `LDS`, for example, loads 16 or 32 bits from the given memory address into the given register (depending on the size of the register), then loads the *next* 16 bits from memory into `DS`. `LES`, `LFS`, `LGS` and `LSS` work in the same way but use the other segment registers.

A.5.135 LEA: Load Effective Address

```
LEA reg16,mem                 ; o16 8D /r            [8086]
LEA reg32,mem                 ; o32 8D /r            [386]
```

`LEA`, despite its syntax, does not access memory. It calculates the effective address specified by its second operand as if it were going to load or store data from it, but instead it stores the calculated address into the register specified by its first operand. This can be used to perform quite complex calculations (e.g. `LEA EAX,[EBX+ECX*4+100]`) in one instruction.

`LEA`, despite being a purely arithmetic instruction which accesses no memory, still requires square brackets around its second operand, as if it were a memory reference.

The size of the calculation is the current *address* size, and the size that the result is stored as is the current *operand* size. If the address and operand size are not the same, then if the addressing mode was 32-bits, the low 16-bits are stored, and if the address was 16-bits, it is zero-extended to 32-bits before storing.

The ModR/M byte (see section A.2.6) can encode a register as source operand, but this is an invalid instruction.

A.5.136 LEAVE: Destroy Stack Frame

```
LEAVE                         ; C9                   [186]
```

`LEAVE` destroys a stack frame of the form created by the `ENTER` instruction (see section A.5.65). It is functionally equivalent to `MOV ESP,EBP` followed by `POP EBP` (or `MOV SP,BP` followed by `POP BP` in 16-bit mode).

A.5.137 LFENCE: Load Fence

```
LFENCE                        ; 0F AE /5        [WILLAMETTE,SSE2]
```

LFENCE performs a serialising operation on all loads from memory that were issued before the LFENCE instruction. This guarantees that all memory reads before the LFENCE instruction are visible before any reads after the LFENCE instruction.

LFENCE is ordered respective to other LFENCE instruction, MFENCE, any memory read and any other serialising instruction (such as CPUID).

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.

LFENCE uses the following ModR/M encoding:

```
Mod (7:6)        = 11B
Reg/Opcode (5:3) = 101B
R/M (2:0)        = 000B
```

All other ModR/M encodings are defined to be reserved, and use of these encodings risks incompatibility with future processors.

See also SFENCE (section A.5.288) and MFENCE (section A.5.151).

A.5.138 LGDT, LIDT, LLDT: Load Descriptor Tables

```
LGDT mem                      ; 0F 01 /2             [286,PRIV]
LIDT mem                      ; 0F 01 /3             [286,PRIV]
LLDT r/m16                    ; 0F 00 /2             [286,PRIV]
```

LGDT and LIDT both take a 6-byte memory area as an operand: they load a 16-bit size limit and a 32-bit linear address from that area (limit word first, then linear address dword) into the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register). The GDT and IDT instructions are the only instructions which directly use *linear* addresses, rather than segment/offset pairs.

LLDT takes a segment selector as an operand. The processor looks up that selector in the GDT and stores the limit and base address given there into the LDTR (local descriptor table register).

See also SGDT, SIDT and SLDT (section A.5.289).

A.5.139 LMSW: Load/Store Machine Status Word

```
LMSW r/m16                    ; 0F 01 /6             [286,PRIV]
```

LMSW loads the bottom four bits of the source operand into the bottom four bits of the CR0 control register (or the Machine Status Word, on 286 processors). See also SMSW (section A.5.296).

A.5.140 LOADALL, LOADALL286: Load Processor State

```
LOADALL                        ; 0F 07                [386,UNDOC]
LOADALL286                     ; 0F 05                [286,UNDOC]
```

This instruction, in its two different-opcode forms, is apparently supported on most 286 processors, some 386 and possibly some 486. The opcode differs between the 286 and the 386.

The function of the instruction is to load all information relating to the state of the processor out of a block of memory: on the 286, this block is located implicitly at absolute address `0x800`, and on the 386 and 486 it is at `[ES:EDI]`.

A.5.141 LODSB, LODSW, LODSD: Load from String

```
LODSB                          ; AC                   [8086]
LODSW                          ; o16 AD               [8086]
LODSD                          ; o32 AD               [386]
```

`LODSB` loads a byte from `[DS:SI]` or `[DS:ESI]` into `AL`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` or `ESI`. The flags are not modified.

The register used is `SI` if the address size is 16 bits, and `ESI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `ES LODSB`).

`LODSW` and `LODSD` work in the same way, but they load a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` - again, the address size chooses which) times. The `REP` alias `REPE`, as well as the differently-encoded `REPNE`, are both allowed as well. They behave in the same way as `REP`.

A.5.141.1 Pseudo-code examples

`a16 LODSB` without segment override and with Direction Flag clear (UP) is equal to

```
MOV AL, BYTE [SI]
LEA SI, [SI + 1]
```

`a32 ES LODSD` with Direction Flag set (DN) is equal to

```
MOV EAX, DWORD [ES:ESI]
LEA ESI, [ESI - 4]
```

A.5.142 LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ: Loop with Counter

```
LOOP imm                       ; E2 rb                [8086]
LOOP imm,CX                    ; a16 E2 rb            [8086]
LOOP imm,ECX                   ; a32 E2 rb            [386]

LOOPE imm                      ; E1 rb                [8086]
LOOPE imm,CX                   ; a16 E1 rb            [8086]
```

```
LOOPE imm,ECX                 ; a32 E1 rb            [386]
LOOPZ imm                     ; E1 rb                [8086]
LOOPZ imm,CX                  ; a16 E1 rb            [8086]
LOOPZ imm,ECX                 ; a32 E1 rb            [386]

LOOPNE imm                    ; E0 rb                [8086]
LOOPNE imm,CX                 ; a16 E0 rb            [8086]
LOOPNE imm,ECX                ; a32 E0 rb            [386]
LOOPNZ imm                    ; E0 rb                [8086]
LOOPNZ imm,CX                 ; a16 E0 rb            [8086]
LOOPNZ imm,ECX                ; a32 E0 rb            [386]
```

`LOOP` decrements its counter register (either `CX` or `ECX` - if one is not specified explicitly, the `BITS` setting dictates which is used) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label. The jump has a range of 128 bytes.

`LOOPE` (or its synonym `LOOPZ`) adds the additional condition that it only jumps if the counter is nonzero *and* the zero flag is set. Similarly, `LOOPNE` (and `LOOPNZ`) jumps only if the counter is nonzero and the zero flag is clear.

A.5.143 `LSL`: Load Segment Limit

```
LSL reg16,r/m16               ; o16 0F 03 /r         [286,PRIV]
LSL reg32,r/m32               ; o32 0F 03 /r         [386,PRIV]
```

`LSL` is given a segment selector in its source (second) operand; it computes the segment limit value by loading the segment limit field from the associated segment descriptor in the `GDT` or `LDT`. (This involves shifting left by 12 bits if the segment limit is page-granular, and not if it is byte-granular; so you end up with a byte limit in either case.) The segment limit obtained is then loaded into the destination (first) operand.

A.5.144 `LTR`: Load Task Register

```
LTR r/m16                     ; 0F 00 /3             [286,PRIV]
```

`LTR` looks up the segment base and limit in the GDT or LDT descriptor specified by the segment selector given as its operand, and loads them into the Task Register.

A.5.145 `MASKMOVDQU`: Byte Mask Write

```
MASKMOVDQU xmm1,xmm2          ; 66 0F F7 /r     [WILLAMETTE,SSE2]
```

`MASKMOVDQU` stores data from xmm1 to the location specified by `ES:(E)DI`. The size of the store depends on the address-size attribute. The most significant bit in each byte of the mask register xmm2 is used to selectively write the data (0 = no write, 1 = write) on a per-byte basis.

A.5.146 `MASKMOVQ`: Byte Mask Write

```
MASKMOVQ mm1,mm2              ; 0F F7 /r        [KATMAI,MMX]
```

`MASKMOVQ` stores data from mm1 to the location specified by `ES:(E)DI`. The size of the store depends on the address-size attribute. The most significant bit in each byte of the mask register mm2 is used to selectively write the data (0 = no write, 1 = write) on a per-byte basis.

A.5.147 `MAXPD`: Return Packed Double-Precision FP Maximum

```
MAXPD xmm1,xmm2/m128          ; 66 0F 5F /r     [WILLAMETTE,SSE2]
```

`MAXPD` performs a SIMD compare of the packed double-precision FP numbers from xmm1 and xmm2/mem, and stores the maximum values of each pair of values in xmm1. If the values being compared are both zeroes, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned).

A.5.148 `MAXPS`: Return Packed Single-Precision FP Maximum

```
MAXPS xmm1,xmm2/m128          ; 0F 5F /r        [KATMAI,SSE]
```

`MAXPS` performs a SIMD compare of the packed single-precision FP numbers from xmm1 and xmm2/mem, and stores the maximum values of each pair of values in xmm1. If the values being compared are both zeroes, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned).

A.5.149 `MAXSD`: Return Scalar Double-Precision FP Maximum

```
MAXSD xmm1,xmm2/m64           ; F2 0F 5F /r     [WILLAMETTE,SSE2]
```

`MAXSD` compares the low-order double-precision FP numbers from xmm1 and xmm2/mem, and stores the maximum value in xmm1. If the values being compared are both zeroes, source2 (xmm2/m64) would be returned. If source2 (xmm2/m64) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned). The high quadword of the destination is left unchanged.

A.5.150 `MAXSS`: Return Scalar Single-Precision FP Maximum

```
MAXSS xmm1,xmm2/m32           ; F3 0F 5F /r     [KATMAI,SSE]
```

`MAXSS` compares the low-order single-precision FP numbers from xmm1 and xmm2/mem, and stores the maximum value in xmm1. If the values being compared are both zeroes, source2 (xmm2/m32) would be returned. If source2 (xmm2/m32) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned). The high three doublewords of the destination are left unchanged.

A.5.151 `MFENCE`: Memory Fence

```
MFENCE                        ; 0F AE /6        [WILLAMETTE,SSE2]
```

`MFENCE` performs a serialising operation on all loads from memory and writes to memory that were issued before the `MFENCE` instruction. This guarantees that all memory reads and writes before the `MFENCE` instruction are completed before any reads and writes after the `MFENCE` instruction.

`MFENCE` is ordered respective to other `MFENCE` instructions, `LFENCE`, `SFENCE`, any memory read and any other serialising instruction (such as `CPUID`).

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.

MFENCE uses the following ModR/M encoding:

```
Mod (7:6)         = 11B
Reg/Opcode (5:3) = 110B
R/M (2:0)         = 000B
```

All other ModR/M encodings are defined to be reserved, and use of these encodings risks incompatibility with future processors.

See also LFENCE (section A.5.137) and SFENCE (section A.5.288).

A.5.152 MINPD: Return Packed Double-Precision FP Minimum

```
MINPD xmm1,xmm2/m128          ; 66 0F 5D /r     [WILLAMETTE,SSE2]
```

MINPD performs a SIMD compare of the packed double-precision FP numbers from xmm1 and xmm2/mem, and stores the minimum values of each pair of values in xmm1. If the values being compared are both zeroes, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned).

A.5.153 MINPS: Return Packed Single-Precision FP Minimum

```
MINPS xmm1,xmm2/m128          ; 0F 5D /r        [KATMAI,SSE]
```

MINPS performs a SIMD compare of the packed single-precision FP numbers from xmm1 and xmm2/mem, and stores the minimum values of each pair of values in xmm1. If the values being compared are both zeroes, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned).

A.5.154 MINSD: Return Scalar Double-Precision FP Minimum

```
MINSD xmm1,xmm2/m64           ; F2 0F 5D /r     [WILLAMETTE,SSE2]
```

MINSD compares the low-order double-precision FP numbers from xmm1 and xmm2/mem, and stores the minimum value in xmm1. If the values being compared are both zeroes, source2 (xmm2/m64) would be returned. If source2 (xmm2/m64) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned). The high quadword of the destination is left unchanged.

A.5.155 MINSS: Return Scalar Single-Precision FP Minimum

```
MINSS xmm1,xmm2/m32           ; F3 0F 5D /r     [KATMAI,SSE]
```

MINSS compares the low-order single-precision FP numbers from xmm1 and xmm2/mem, and stores the minimum value in xmm1. If the values being compared are both zeroes, source2 (xmm2/m32) would be returned. If source2 (xmm2/m32) is an SNaN, this SNaN is forwarded unchanged to the destination (i.e., a QNaN version of the SNaN is not returned). The high three doublewords of the destination are left unchanged.

A.5.156 `MOV`: Move Data

```
MOV r/m8,reg8                  ; 88 /r                [8086]
MOV r/m16,reg16                ; o16 89 /r            [8086]
MOV r/m32,reg32                ; o32 89 /r            [386]
MOV reg8,r/m8                  ; 8A /r                [8086]
MOV reg16,r/m16                ; o16 8B /r            [8086]
MOV reg32,r/m32                ; o32 8B /r            [386]

MOV reg8,imm8                  ; B0+r ib              [8086]
MOV reg16,imm16                ; o16 B8+r iw          [8086]
MOV reg32,imm32                ; o32 B8+r id          [386]
MOV r/m8,imm8                  ; C6 /0 ib             [8086]
MOV r/m16,imm16                ; o16 C7 /0 iw         [8086]
MOV r/m32,imm32                ; o32 C7 /0 id         [386]

MOV AL,memoffs8                ; A0 ow/od             [8086]
MOV AX,memoffs16               ; o16 A1 ow/od         [8086]
MOV EAX,memoffs32              ; o32 A1 ow/od         [386]
MOV memoffs8,AL                ; A2 ow/od             [8086]
MOV memoffs16,AX               ; o16 A3 ow/od         [8086]
MOV memoffs32,EAX              ; o32 A3 ow/od         [386]

MOV r/m16,segreg               ; o16 8C /r            [8086]
MOV r/m32,segreg               ; o32 8C /r            [386]
MOV segreg,r/m16               ; o16 8E /r            [8086]
MOV segreg,r/m32               ; o32 8E /r            [386]

MOV reg32,CR0/2/3/4            ; 0F 20 /r             [386]
MOV reg32,DR0/1/2/3/6/7        ; 0F 21 /r             [386]
MOV reg32,TR3/4/5/6/7          ; 0F 24 /r             [386]
MOV CR0/2/3/4,reg32            ; 0F 22 /r             [386]
MOV DR0/1/2/3/6/7,reg32        ; 0F 23 /r             [386]
MOV TR3/4/5/6/7,reg32          ; 0F 26 /r             [386]
```

`MOV` copies the contents of its source (second) operand into its destination (first) operand.

In all forms of the `MOV` instruction, the two operands are the same size, except for moving between a segment register and an `r/m32` operand. These instructions are treated exactly like the corresponding 16-bit equivalent (so that, for example, `MOV DS,EAX` functions identically to `MOV DS,AX` but saves a prefix when in 32-bit mode), except that when a segment register is moved into a 32-bit destination, the top two bytes of the result are undefined.

`MOV` may not use `CS` as a destination. However, this can be encoded, which is silently accepted by current versions of NASM, and is decoded by NDISASM.

`CR4` is only a supported register on the Pentium and above.

Test registers are supported on 386/486 processors and on some non-Intel Pentium class processors.

A.5.157 `MOVAPD`: Move Aligned Packed Double-Precision FP Values

```
MOVAPD xmm1,xmm2/mem128        ; 66 0F 28 /r     [WILLAMETTE,SSE2]
```

```
MOVAPD xmm1/mem128,xmm2       ; 66 0F 29 /r     [WILLAMETTE,SSE2]
```

MOVAPD moves a double quadword containing 2 packed double-precision FP values from the source operand to the destination. When the source or destination operand is a memory location, it must be aligned on a 16-byte boundary.

To move data in and out of memory locations that are not known to be on 16-byte boundaries, use the MOVUPD instruction (section A.5.182).

A.5.158 MOVAPS: Move Aligned Packed Single-Precision FP Values

```
MOVAPS xmm1,xmm2/mem128       ; 0F 28 /r        [KATMAI,SSE]
MOVAPS xmm1/mem128,xmm2       ; 0F 29 /r        [KATMAI,SSE]
```

MOVAPS moves a double quadword containing 4 packed single-precision FP values from the source operand to the destination. When the source or destination operand is a memory location, it must be aligned on a 16-byte boundary.

To move data in and out of memory locations that are not known to be on 16-byte boundaries, use the MOVUPS instruction (section A.5.183).

A.5.159 MOVD: Move Doubleword to/from MMX Register

```
MOVD mm,r/m32                 ; 0F 6E /r             [PENT,MMX]
MOVD r/m32,mm                 ; 0F 7E /r             [PENT,MMX]
MOVD xmm,r/m32                ; 66 0F 6E /r     [WILLAMETTE,SSE2]
MOVD r/m32,xmm                ; 66 0F 7E /r     [WILLAMETTE,SSE2]
```

MOVD copies 32 bits from its source (second) operand into its destination (first) operand. When the destination is a 64-bit MMX register or a 128-bit XMM register, the input value is zero-extended to fill the destination register.

A.5.160 MOVDQ2Q: Move Quadword from XMM to MMX register.

```
MOVDQ2Q mm,xmm                ; F2 OF D6 /r     [WILLAMETTE,SSE2]
```

MOVDQ2Q moves the low quadword from the source operand to the destination operand.

A.5.161 MOVDQA: Move Aligned Double Quadword

```
MOVDQA xmm1,xmm2/m128         ; 66 OF 6F /r     [WILLAMETTE,SSE2]
MOVDQA xmm1/m128,xmm2         ; 66 OF 7F /r     [WILLAMETTE,SSE2]
```

MOVDQA moves a double quadword from the source operand to the destination operand. When the source or destination operand is a memory location, it must be aligned to a 16-byte boundary.

To move a double quadword to or from unaligned memory locations, use the MOVDQU instruction (section A.5.162).

A.5.162 MOVDQU: Move Unaligned Double Quadword

```
MOVDQU xmm1,xmm2/m128         ; F3 OF 6F /r     [WILLAMETTE,SSE2]
MOVDQU xmm1/m128,xmm2         ; F3 OF 7F /r     [WILLAMETTE,SSE2]
```

MOVDQU moves a double quadword from the source operand to the destination operand. When the source or destination operand is a memory location, the memory may be unaligned.

To move a double quadword to or from known aligned memory locations, use the MOVDQA instruction (section A.5.161).

A.5.163 MOVHLPS: Move Packed Single-Precision FP High to Low

```
MOVHLPS xmm1,xmm2               ; OF 12 /r        [KATMAI,SSE]
```

MOVHLPS moves the two packed single-precision FP values from the high quadword of the source register xmm2 to the low quadword of the destination register, xmm2. The upper quadword of xmm1 is left unchanged.

The operation of this instruction is:

```
   dst[0-63]   := src[64-127],
   dst[64-127] remains unchanged.
```

A.5.164 MOVHPD: Move High Packed Double-Precision FP

```
MOVHPD xmm,m64                 ; 66 OF 16 /r      [WILLAMETTE,SSE2]
MOVHPD m64,xmm                 ; 66 OF 17 /r      [WILLAMETTE,SSE2]
```

MOVHPD moves a double-precision FP value between the source and destination operands. One of the operands is a 64-bit memory location, the other is the high quadword of an XMM register.

The operation of this instruction is:

```
   mem[0-63]   := xmm[64-127];
```

or

```
   xmm[0-63]   remains unchanged;
   xmm[64-127] := mem[0-63].
```

A.5.165 MOVHPS: Move High Packed Single-Precision FP

```
MOVHPS xmm,m64                 ; 0F 16 /r         [KATMAI,SSE]
MOVHPS m64,xmm                 ; 0F 17 /r         [KATMAI,SSE]
```

MOVHPS moves two packed single-precision FP values between the source and destination operands. One of the operands is a 64-bit memory location, the other is the high quadword of an XMM register.

The operation of this instruction is:

```
   mem[0-63]   := xmm[64-127];
```

or

```
   xmm[0-63]   remains unchanged;
   xmm[64-127] := mem[0-63].
```

A.5.166 MOVLHPS: Move Packed Single-Precision FP Low to High

```
MOVLHPS xmm1,xmm2               ; OF 16 /r         [KATMAI,SSE]
```

MOVLHPS moves the two packed single-precision FP values from the low quadword of the source register xmm2 to the high quadword of the destination register, xmm2. The low quadword of xmm1 is left unchanged.

The operation of this instruction is:

```
dst[0-63]   remains unchanged;
dst[64-127] := src[0-63].
```

A.5.167 MOVLPD: Move Low Packed Double-Precision FP

```
MOVLPD xmm,m64                ; 66 OF 12 /r     [WILLAMETTE,SSE2]
MOVLPD m64,xmm                ; 66 OF 13 /r     [WILLAMETTE,SSE2]
```

MOVLPD moves a double-precision FP value between the source and destination operands. One of the operands is a 64-bit memory location, the other is the low quadword of an XMM register.

The operation of this instruction is:

```
mem(0-63)   := xmm(0-63);
```

or

```
xmm(0-63)   := mem(0-63);
xmm(64-127) remains unchanged.
```

A.5.168 MOVLPS: Move Low Packed Single-Precision FP

```
MOVLPS xmm,m64                ; OF 12 /r        [KATMAI,SSE]
MOVLPS m64,xmm                ; OF 13 /r        [KATMAI,SSE]
```

MOVLPS moves two packed single-precision FP values between the source and destination operands. One of the operands is a 64-bit memory location, the other is the low quadword of an XMM register.

The operation of this instruction is:

```
mem(0-63)   := xmm(0-63);
```

or

```
xmm(0-63)   := mem(0-63);
xmm(64-127) remains unchanged.
```

A.5.169 MOVMSKPD: Extract Packed Double-Precision FP Sign Mask

```
MOVMSKPD reg32,xmm              ; 66 0F 50 /r   [WILLAMETTE,SSE2]
```

MOVMSKPD inserts a 2-bit mask in r32, formed of the most significant bits of each double-precision FP number of the source operand.

A.5.170 MOVMSKPS: Extract Packed Single-Precision FP Sign Mask

```
MOVMSKPS reg32,xmm              ; 0F 50 /r      [KATMAI,SSE]
```

MOVMSKPS inserts a 4-bit mask in r32, formed of the most significant bits of each single-precision FP number of the source operand.

A.5.171 `MOVNTDQ`: Move Double Quadword Non Temporal

```
MOVNTDQ m128,xmm              ; 66 0F E7 /r     [WILLAMETTE,SSE2]
```

`MOVNTDQ` moves the double quadword from the `XMM` source register to the destination memory location, using a non-temporal hint. This store instruction minimizes cache pollution.

A.5.172 `MOVNTI`: Move Doubleword Non Temporal

```
MOVNTI m32,reg32              ; 0F C3 /r        [WILLAMETTE,SSE2]
```

`MOVNTI` moves the doubleword in the source register to the destination memory location, using a non-temporal hint. This store instruction minimizes cache pollution.

A.5.173 `MOVNTPD`: Move Aligned Four Packed Single-Precision FP Values Non Temporal

```
MOVNTPD m128,xmm              ; 66 0F 2B /r     [WILLAMETTE,SSE2]
```

`MOVNTPD` moves the double quadword from the `XMM` source register to the destination memory location, using a non-temporal hint. This store instruction minimizes cache pollution. The memory location must be aligned to a 16-byte boundary.

A.5.174 `MOVNTPS`: Move Aligned Four Packed Single-Precision FP Values Non Temporal

```
MOVNTPS m128,xmm              ; 0F 2B /r        [KATMAI,SSE]
```

`MOVNTPS` moves the double quadword from the `XMM` source register to the destination memory location, using a non-temporal hint. This store instruction minimizes cache pollution. The memory location must be aligned to a 16-byte boundary.

A.5.175 `MOVNTQ`: Move Quadword Non Temporal

```
MOVNTQ m64,mm                 ; 0F E7 /r        [KATMAI,MMX]
```

`MOVNTQ` moves the quadword in the `MMX` source register to the destination memory location, using a non-temporal hint. This store instruction minimizes cache pollution.

A.5.176 `MOVQ`: Move Quadword to/from MMX Register

```
MOVQ mm1,mm2/m64               ; 0F 6F /r             [PENT,MMX]
MOVQ mm1/m64,mm2               ; 0F 7F /r             [PENT,MMX]

MOVQ xmm1,xmm2/m64             ; F3 0F 7E /r    [WILLAMETTE,SSE2]
MOVQ xmm1/m64,xmm2             ; 66 0F D6 /r    [WILLAMETTE,SSE2]
```

`MOVQ` copies 64 bits from its source (second) operand into its destination (first) operand. When the source is an `XMM` register, the low quadword is moved. When the destination is an `XMM` register, the destination is the low quadword, and the high quadword is cleared.

A.5.177 `MOVQ2DQ`: Move Quadword from MMX to XMM register.

```
MOVQ2DQ xmm,mm                ; F3 OF D6 /r     [WILLAMETTE,SSE2]
```

MOVQ2DQ moves the quadword from the source operand to the low quadword of the destination operand, and clears the high quadword.

A.5.178 MOVSB, MOVSW, MOVSD: Move String

```
MOVSB                          ; A4                     [8086]
MOVSW                          ; o16 A5                 [8086]
MOVSD                          ; o32 A5                 [386]
```

MOVSB copies the byte at [DS:SI] or [DS:ESI] to [ES:DI] or [ES:EDI]. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) SI and DI (or ESI and EDI). The flags are not modified.

The registers used are SI and DI if the address size is 16 bits, and ESI and EDI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

The segment register used to load from [SI] or [ESI] can be overridden by using a segment register name as a prefix (for example, ES MOVSB). The use of ES for the store to [DI] or [EDI] cannot be overridden.

MOVSW and MOVSD work in the same way, but they copy a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REP prefix may be used to repeat the instruction CX (or ECX - again, the address size chooses which) times. The REP alias REPE, as well as the differently-encoded REPNE, are both allowed as well. They behave in the same way as REP.

Note that MOVSB is equivalent to LODSB followed by STOSB, except it doesn't write the accumulator register (AL).

A.5.178.1 Pseudo-code examples

a16 MOVSB without segment override and with Direction Flag clear (UP) is equal to

```
MOV BYTE [ES:DI], BYTE [SI]
LEA SI, [SI + 1]
LEA DI, [DI + 1]
```

a16 REP MOVSW without segment override and with Direction Flag clear (UP) is equal to

```
JCXZ @FF
@@:
MOV WORD [ES:DI], WORD [SI]
LEA SI, [SI + 2]
LEA DI, [DI + 2]
a16 LOOP @B
@@:
```

a32 ES MOVSD with Direction Flag set (DN) is equal to

```
MOV DWORD [ES:EDI], DWORD [ES:ESI]
LEA ESI, [ESI - 4]
LEA EDI, [EDI - 4]
```

A.5.179 `MOVSD`: Move Scalar Double-Precision FP Value

```
MOVSD xmm1,xmm2/m64           ; F2 0F 10 /r     [WILLAMETTE,SSE2]
MOVSD xmm1/m64,xmm2           ; F2 0F 11 /r     [WILLAMETTE,SSE2]
```

`MOVSD` moves a double-precision FP value from the source operand to the destination operand. When the source or destination is a register, the low-order FP value is read or written.

A.5.180 `MOVSS`: Move Scalar Single-Precision FP Value

```
MOVSS xmm1,xmm2/m32           ; F3 0F 10 /r     [KATMAI,SSE]
MOVSS xmm1/m32,xmm2           ; F3 0F 11 /r     [KATMAI,SSE]
```

`MOVSS` moves a single-precision FP value from the source operand to the destination operand. When the source or destination is a register, the low-order FP value is read or written.

A.5.181 `MOVSX`, `MOVZX`: Move Data with Sign or Zero Extend

```
MOVSX reg16,r/m8              ; o16 0F BE /r         [386]
MOVSX reg32,r/m8              ; o32 0F BE /r         [386]
MOVSX reg32,r/m16             ; o32 0F BF /r         [386]

MOVZX reg16,r/m8              ; o16 0F B6 /r         [386]
MOVZX reg32,r/m8              ; o32 0F B6 /r         [386]
MOVZX reg32,r/m16             ; o32 0F B7 /r         [386]
```

`MOVSX` sign-extends its source (second) operand to the length of its destination (first) operand, and copies the result into the destination operand. `MOVZX` does the same, but zero-extends rather than sign-extending.

A.5.182 `MOVUPD`: Move Unaligned Packed Double-Precision FP Values

```
MOVUPD xmm1,xmm2/mem128       ; 66 0F 10 /r     [WILLAMETTE,SSE2]
MOVUPD xmm1/mem128,xmm2       ; 66 0F 11 /r     [WILLAMETTE,SSE2]
```

`MOVUPD` moves a double quadword containing 2 packed double-precision FP values from the source operand to the destination. This instruction makes no assumptions about alignment of memory operands.

To move data in and out of memory locations that are known to be on 16-byte boundaries, use the `MOVAPD` instruction (section A.5.157).

A.5.183 `MOVUPS`: Move Unaligned Packed Single-Precision FP Values

```
MOVUPS xmm1,xmm2/mem128       ; 0F 10 /r        [KATMAI,SSE]
MOVUPS xmm1/mem128,xmm2       ; 0F 11 /r        [KATMAI,SSE]
```

`MOVUPS` moves a double quadword containing 4 packed single-precision FP values from the source operand to the destination. This instruction makes no assumptions about alignment of memory operands.

To move data in and out of memory locations that are known to be on 16-byte boundaries, use the `MOVAPS` instruction (section A.5.158).

A.5.184 MUL: Unsigned Integer Multiply

```
MUL r/m8                      ; F6 /4                [8086]
MUL r/m16                     ; o16 F7 /4            [8086]
MUL r/m32                     ; o32 F7 /4            [386]
```

MUL performs unsigned integer multiplication. The other operand to the multiplication, and the destination operand, are implicit, in the following way:

- For MUL r/m8, AL is multiplied by the given operand; the product is stored in AX.
- For MUL r/m16, AX is multiplied by the given operand; the product is stored in DX:AX.
- For MUL r/m32, EAX is multiplied by the given operand; the product is stored in EDX:EAX.

Signed integer multiplication is performed by the IMUL instruction: see section A.5.118.

A.5.185 MULPD: Packed Single-FP Multiply

```
MULPD xmm1,xmm2/mem128        ; 66 0F 59 /r     [WILLAMETTE,SSE2]
```

MULPD performs a SIMD multiply of the packed double-precision FP values in both operands, and stores the results in the destination register.

A.5.186 MULPS: Packed Single-FP Multiply

```
MULPS xmm1,xmm2/mem128        ; 0F 59 /r        [KATMAI,SSE]
```

MULPS performs a SIMD multiply of the packed single-precision FP values in both operands, and stores the results in the destination register.

A.5.187 MULSD: Scalar Single-FP Multiply

```
MULSD xmm1,xmm2/mem32         ; F2 0F 59 /r     [WILLAMETTE,SSE2]
```

MULSD multiplies the lowest double-precision FP values of both operands, and stores the result in the low quadword of xmm1.

A.5.188 MULSS: Scalar Single-FP Multiply

```
MULSS xmm1,xmm2/mem32         ; F3 0F 59 /r     [KATMAI,SSE]
```

MULSS multiplies the lowest single-precision FP values of both operands, and stores the result in the low doubleword of xmm1.

A.5.189 NEG, NOT: Two's and Ones' Complement

```
NEG r/m8                      ; F6 /3                [8086]
NEG r/m16                     ; o16 F7 /3            [8086]
NEG r/m32                     ; o32 F7 /3            [386]

NOT r/m8                      ; F6 /2                [8086]
NOT r/m16                     ; o16 F7 /2            [8086]
NOT r/m32                     ; o32 F7 /2            [386]
```

NEG replaces the contents of its operand by the two's complement negation (invert all the bits and then add one) of the original value. NOT, similarly, performs ones' complement (inverts all the bits).

NOT does not modify any flags.

NEG sets flags like as if running a subtraction where the operand is subtracted from zero. That means the Carry Flag is cleared by NEG if the operand was zero, and set otherwise. The Zero Flag is set by NEG according to whether the result is zero. If the NEG operand was zero it means the result is also zero, and vice versa.

A.5.190 NOP: No Operation

```
NOP                           ; 90                     [8086]
```

NOP performs no operation. Its opcode is the same as that generated by XCHG AX,AX or XCHG EAX,EAX (depending on the processor mode; see section A.5.333).

A.5.191 OR: Bitwise OR

```
OR r/m8,reg8                  ; 08 /r                  [8086]
OR r/m16,reg16                ; o16 09 /r              [8086]
OR r/m32,reg32                ; o32 09 /r              [386]

OR reg8,r/m8                  ; 0A /r                  [8086]
OR reg16,r/m16                ; o16 0B /r              [8086]
OR reg32,r/m32                ; o32 0B /r              [386]

OR r/m8,imm8                  ; 80 /1 ib               [8086]
OR r/m16,imm16                ; o16 81 /1 iw           [8086]
OR r/m32,imm32                ; o32 81 /1 id           [386]

OR r/m16,imm8                 ; o16 83 /1 ib           [8086]
OR r/m32,imm8                 ; o32 83 /1 ib           [386]

OR AL,imm8                    ; 0C ib                  [8086]
OR AX,imm16                   ; o16 0D iw              [8086]
OR EAX,imm32                  ; o32 0D id              [386]
```

OR performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The BYTE qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the STRICT WORD or STRICT DWORD qualifier.

The Carry Flag is cleared by OR. The Zero Flag is set according to whether the result is zero.

The MMX instruction POR (see section A.5.247) performs the same operation on the 64-bit MMX registers.

A.5.192 **ORPD**: Bit-wise Logical OR of Double-Precision FP Data

```
ORPD xmm1,xmm2/m128            ; 66 0F 56 /r     [WILLAMETTE,SSE2]
```

ORPD return a bit-wise logical OR between xmm1 and xmm2/mem, and stores the result in xmm1. If the source operand is a memory location, it must be aligned to a 16-byte boundary.

A.5.193 **ORPS**: Bit-wise Logical OR of Single-Precision FP Data

```
ORPS xmm1,xmm2/m128            ; 0F 56 /r        [KATMAI,SSE]
```

ORPS return a bit-wise logical OR between xmm1 and xmm2/mem, and stores the result in xmm1. If the source operand is a memory location, it must be aligned to a 16-byte boundary.

A.5.194 **OUT**: Output Data to I/O Port

```
OUT imm8,AL                    ; E6 ib                [8086]
OUT imm8,AX                    ; o16 E7 ib            [8086]
OUT imm8,EAX                   ; o32 E7 ib            [386]
OUT DX,AL                      ; EE                   [8086]
OUT DX,AX                      ; o16 EF               [8086]
OUT DX,EAX                     ; o32 EF               [386]
```

OUT writes the contents of the given source register to the specified I/O port. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in DX. See also IN (section A.5.119).

A.5.195 **OUTSB**, **OUTSW**, **OUTSD**: Output String to I/O Port

```
OUTSB                          ; 6E                   [186]
OUTSW                          ; o16 6F               [186]
OUTSD                          ; o32 6F               [386]
```

OUTSB loads a byte from [DS:SI] or [DS:ESI] and writes it to the I/O port specified in DX. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) SI or ESI. The flags are not modified.

The register used is SI if the address size is 16 bits, and ESI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

The segment register used to load from [SI] or [ESI] can be overridden by using a segment register name as a prefix (for example, ES OUTSB).

OUTSW and OUTSD work in the same way, but they output a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REP prefix may be used to repeat the instruction CX (or ECX - again, the address size chooses which) times. The REP alias REPE, as well as the differently-encoded REPNE, are both allowed as well. They behave in the same way as REP.

Note that OUTSB is equivalent to LODSB followed by OUT DX,AL, except it doesn't write the accumulator register (AL).

A.5.195.1 Pseudo-code examples

`a16 OUTSB` without segment override and with Direction Flag clear (UP) is equal to

```
OUT DX, BYTE [SI]
LEA SI, [SI + 1]
```

`a16 REP OUTSW` without segment override and with Direction Flag clear (UP) is equal to

```
JCXZ @FF
@@:
OUT DX, WORD [SI]
LEA SI, [SI + 2]
a16 LOOP @B
@@:
```

`a32 ES OUTSD` with Direction Flag set (DN) is equal to

```
OUT DX, DWORD [ES:ESI]
LEA ESI, [ESI - 4]
```

A.5.196 PACKSSDW, PACKSSWB, PACKUSWB: Pack Data

```
PACKSSDW mm1,mm2/m64          ; 0F 6B /r              [PENT,MMX]
PACKSSWB mm1,mm2/m64          ; 0F 63 /r              [PENT,MMX]
PACKUSWB mm1,mm2/m64          ; 0F 67 /r              [PENT,MMX]

PACKSSDW xmm1,xmm2/m128       ; 66 0F 6B /r     [WILLAMETTE,SSE2]
PACKSSWB xmm1,xmm2/m128       ; 66 0F 63 /r     [WILLAMETTE,SSE2]
PACKUSWB xmm1,xmm2/m128       ; 66 0F 67 /r     [WILLAMETTE,SSE2]
```

All these instructions start by combining the source and destination operands, and then splitting the result in smaller sections which it then packs into the destination register. The `MMX` versions pack two 64-bit operands into one 64-bit register, while the `SSE` versions pack two 128-bit operands into one 128-bit register.

- `PACKSSWB` splits the combined value into words, and then reduces the words to bytes, using signed saturation. It then packs the bytes into the destination register in the same order the words were in.

- `PACKSSDW` performs the same operation as `PACKSSWB`, except that it reduces doublewords to words, then packs them into the destination register.

- `PACKUSWB` performs the same operation as `PACKSSWB`, except that it uses unsigned saturation when reducing the size of the elements.

To perform signed saturation on a number, it is replaced by the largest signed number (`7FFFh` or `7Fh`) that *will* fit, and if it is too small it is replaced by the smallest signed number (`8000h` or `80h`) that will fit. To perform unsigned saturation, the input is treated as unsigned, and the input is replaced by the largest unsigned number that will fit.

A.5.197 PADDB, PADDW, PADDD: Add Packed Integers

```
PADDB mm1,mm2/m64             ; 0F FC /r           [PENT,MMX]
PADDW mm1,mm2/m64             ; 0F FD /r           [PENT,MMX]
PADDD mm1,mm2/m64             ; 0F FE /r           [PENT,MMX]

PADDB xmm1,xmm2/m128          ; 66 0F FC /r   [WILLAMETTE,SSE2]
PADDW xmm1,xmm2/m128          ; 66 0F FD /r   [WILLAMETTE,SSE2]
PADDD xmm1,xmm2/m128          ; 66 0F FE /r   [WILLAMETTE,SSE2]
```

`PADDx` performs packed addition of the two operands, storing the result in the destination (first) operand.

- `PADDB` treats the operands as packed bytes, and adds each byte individually;
- `PADDW` treats the operands as packed words;
- `PADDD` treats its operands as packed doublewords.

When an individual result is too large to fit in its destination, it is wrapped around and the low bits are stored, with the carry bit discarded.

A.5.198 PADDQ: Add Packed Quadword Integers

```
PADDQ mm1,mm2/m64             ; 0F D4 /r           [PENT,MMX]

PADDQ xmm1,xmm2/m128          ; 66 0F D4 /r   [WILLAMETTE,SSE2]
```

`PADDQ` adds the quadwords in the source and destination operands, and stores the result in the destination register.

When an individual result is too large to fit in its destination, it is wrapped around and the low bits are stored, with the carry bit discarded.

A.5.199 PADDSB, PADDSW: Add Packed Signed Integers With Saturation

```
PADDSB mm1,mm2/m64            ; 0F EC /r           [PENT,MMX]
PADDSW mm1,mm2/m64            ; 0F ED /r           [PENT,MMX]

PADDSB xmm1,xmm2/m128         ; 66 0F EC /r   [WILLAMETTE,SSE2]
PADDSW xmm1,xmm2/m128         ; 66 0F ED /r   [WILLAMETTE,SSE2]
```

`PADDSx` performs packed addition of the two operands, storing the result in the destination (first) operand. `PADDSB` treats the operands as packed bytes, and adds each byte individually; and `PADDSW` treats the operands as packed words.

When an individual result is too large to fit in its destination, a saturated value is stored. The resulting value is the value with the largest magnitude of the same sign as the result which will fit in the available space.

A.5.200 PADDSIW: MMX Packed Addition to Implicit Destination

```
PADDSIW mmxreg,r/m64          ; 0F 51 /r          [CYRIX,MMX]
```

`PADDSIW`, specific to the Cyrix extensions to the MMX instruction set, performs the same function as PADDSW, except that the result is placed in an implied register.

To work out the implied register, invert the lowest bit in the register number. So PADDSIW MM0,MM2 would put the result in MM1, but PADDSIW MM1,MM2 would put the result in MM0.

A.5.201 PADDUSB, PADDUSW: Add Packed Unsigned Integers With Saturation

```
PADDUSB mm1,mm2/m64              ; 0F DC /r             [PENT,MMX]
PADDUSW mm1,mm2/m64              ; 0F DD /r             [PENT,MMX]

PADDUSB xmm1,xmm2/m128            ; 66 0F DC /r    [WILLAMETTE,SSE2]
PADDUSW xmm1,xmm2/m128            ; 66 0F DD /r    [WILLAMETTE,SSE2]
```

PADDUSx performs packed addition of the two operands, storing the result in the destination (first) operand. PADDUSB treats the operands as packed bytes, and adds each byte individually; and PADDUSW treats the operands as packed words.

When an individual result is too large to fit in its destination, a saturated value is stored. The resulting value is the maximum value that will fit in the available space.

A.5.202 PAND, PANDN: MMX Bitwise AND and AND-NOT

```
PAND mm1,mm2/m64                 ; 0F DB /r             [PENT,MMX]
PANDN mm1,mm2/m64                ; 0F DF /r             [PENT,MMX]

PAND xmm1,xmm2/m128              ; 66 0F DB /r     [WILLAMETTE,SSE2]
PANDN xmm1,xmm2/m128             ; 66 0F DF /r     [WILLAMETTE,SSE2]
```

PAND performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand.

PANDN performs the same operation, but performs a ones' complement operation on the destination (first) operand first.

A.5.203 PAUSE: Spin Loop Hint

```
PAUSE                            ; F3 90           [WILLAMETTE,SSE2]
```

PAUSE provides a hint to the processor that the following code is a spin loop. This improves processor performance by bypassing possible memory order violations. On older processors, this instruction operates as a NOP.

A.5.204 PAVEB: MMX Packed Average

```
PAVEB mmxreg,r/m64               ; 0F 50 /r              [CYRIX,MMX]
```

PAVEB, specific to the Cyrix MMX extensions, treats its two operands as vectors of eight unsigned bytes, and calculates the average of the corresponding bytes in the operands. The resulting vector of eight averages is stored in the first operand.

This opcode maps to MOVMSKPS r32, xmm on processors that support the SSE instruction set.

A.5.205 PAVGB PAVGW: Average Packed Integers

```
PAVGB mm1,mm2/m64             ; 0F E0 /r         [KATMAI,MMX]
PAVGW mm1,mm2/m64             ; 0F E3 /r         [KATMAI,MMX,SM]

PAVGB xmm1,xmm2/m128          ; 66 0F E0 /r      [WILLAMETTE,SSE2]
PAVGW xmm1,xmm2/m128          ; 66 0F E3 /r      [WILLAMETTE,SSE2]
```

PAVGB and PAVGW add the unsigned data elements of the source operand to the unsigned data elements of the destination register, then adds 1 to the temporary results. The results of the add are then each independently right-shifted by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

- PAVGB operates on packed unsigned bytes, and
- PAVGW operates on packed unsigned words.

A.5.206 PAVGUSB: Average of unsigned packed 8-bit values

```
PAVGUSB mm1,mm2/m64           ; 0F 0F /r BF          [PENT,3DNOW]
```

PAVGUSB adds the unsigned data elements of the source operand to the unsigned data elements of the destination register, then adds 1 to the temporary results. The results of the add are then each independently right-shifted by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

This instruction performs exactly the same operations as the PAVGB MMX instruction (section A.5.205).

A.5.207 PCMPxx: Compare Packed Integers.

```
PCMPEQB mm1,mm2/m64           ; 0F 74 /r             [PENT,MMX]
PCMPEQW mm1,mm2/m64           ; 0F 75 /r             [PENT,MMX]
PCMPEQD mm1,mm2/m64           ; 0F 76 /r             [PENT,MMX]

PCMPGTB mm1,mm2/m64           ; 0F 64 /r             [PENT,MMX]
PCMPGTW mm1,mm2/m64           ; 0F 65 /r             [PENT,MMX]
PCMPGTD mm1,mm2/m64           ; 0F 66 /r             [PENT,MMX]

PCMPEQB xmm1,xmm2/m128        ; 66 0F 74 /r      [WILLAMETTE,SSE2]
PCMPEQW xmm1,xmm2/m128        ; 66 0F 75 /r      [WILLAMETTE,SSE2]
PCMPEQD xmm1,xmm2/m128        ; 66 0F 76 /r      [WILLAMETTE,SSE2]

PCMPGTB xmm1,xmm2/m128        ; 66 0F 64 /r      [WILLAMETTE,SSE2]
PCMPGTW xmm1,xmm2/m128        ; 66 0F 65 /r      [WILLAMETTE,SSE2]
PCMPGTD xmm1,xmm2/m128        ; 66 0F 66 /r      [WILLAMETTE,SSE2]
```

The PCMPxx instructions all treat their operands as vectors of bytes, words, or doublewords; corresponding elements of the source and destination are compared, and the corresponding element of the destination (first) operand is set to all zeros or all ones depending on the result of the comparison.

- PCMPxxB treats the operands as vectors of bytes;
- PCMPxxW treats the operands as vectors of words;

- PCMPxxD treats the operands as vectors of doublewords;
- PCMPEQx sets the corresponding element of the destination operand to all ones if the two elements compared are equal;
- PCMPGTx sets the destination element to all ones if the element of the first (destination) operand is greater (treated as a signed integer) than that of the second (source) operand.

A.5.208 PDISTIB: MMX Packed Distance and Accumulate with Implied Register

```
PDISTIB mm,m64                ; 0F 54 /r             [CYRIX,MMX]
```

PDISTIB, specific to the Cyrix MMX extensions, treats its two input operands as vectors of eight unsigned bytes. For each byte position, it finds the absolute difference between the bytes in that position in the two input operands, and adds that value to the byte in the same position in the implied output register. The addition is saturated to an unsigned byte in the same way as PADDUSB.

To work out the implied register, invert the lowest bit in the register number. So PDISTIB MM0,M64 would put the result in MM1, but PDISTIB MM1,M64 would put the result in MM0.

Note that PDISTIB cannot take a register as its second source operand.

Operation:

```
dstI[0-7]     := dstI[0-7]   + ABS(src0[0-7] - src1[0-7]),
dstI[8-15]    := dstI[8-15]  + ABS(src0[8-15] - src1[8-15]),
.......
.......
dstI[56-63]   := dstI[56-63] + ABS(src0[56-63] - src1[56-63]).
```

A.5.209 PEXTRW: Extract Word

```
PEXTRW reg32,mm,imm8          ; 0F C5 /r ib     [KATMAI,MMX]
PEXTRW reg32,xmm,imm8         ; 66 0F C5 /r ib  [WILLAMETTE,SSE2]
```

PEXTRW moves the word in the source register (second operand) that is pointed to by the count operand (third operand), into the lower half of a 32-bit general purpose register. The upper half of the register is cleared to all 0s.

When the source operand is an MMX register, the two least significant bits of the count specify the source word. When it is an SSE register, the three least significant bits specify the word location.

A.5.210 PF2ID: Packed Single-Precision FP to Integer Convert

```
PF2ID mm1,mm2/m64             ; 0F 0F /r 1D          [PENT,3DNOW]
```

PF2ID converts two single-precision FP values in the source operand to signed 32-bit integers, using truncation, and stores them in the destination operand. Source values that are outside the range supported by the destination are saturated to the largest absolute value of the same sign.

A.5.211 `PF2IW`: Packed Single-Precision FP to Integer Word Convert

```
PF2IW mm1,mm2/m64              ; 0F 0F /r 1C          [PENT,3DNOW]
```

`PF2IW` converts two single-precision FP values in the source operand to signed 16-bit integers, using truncation, and stores them in the destination operand. Source values that are outside the range supported by the destination are saturated to the largest absolute value of the same sign.

- In the K6-2 and K6-III, the 16-bit value is zero-extended to 32-bits before storing.
- In the K6-2+, K6-III+ and Athlon processors, the value is sign-extended to 32-bits before storing.

A.5.212 `PFACC`: Packed Single-Precision FP Accumulate

```
PFACC mm1,mm2/m64              ; 0F 0F /r AE          [PENT,3DNOW]
```

`PFACC` adds the two single-precision FP values from the destination operand together, then adds the two single-precision FP values from the source operand, and places the results in the low and high doublewords of the destination operand.

The operation is:

```
dst[0-31]   := dst[0-31] + dst[32-63],
dst[32-63]  := src[0-31] + src[32-63].
```

A.5.213 `PFADD`: Packed Single-Precision FP Addition

```
PFADD mm1,mm2/m64              ; 0F 0F /r 9E          [PENT,3DNOW]
```

`PFADD` performs addition on each of two packed single-precision FP value pairs.

```
dst[0-31]   := dst[0-31]  + src[0-31],
dst[32-63]  := dst[32-63] + src[32-63].
```

A.5.214 `PFCMPxx`: Packed Single-Precision FP Compare

```
PFCMPEQ mm1,mm2/m64            ; 0F 0F /r B0          [PENT,3DNOW]
PFCMPGE mm1,mm2/m64            ; 0F 0F /r 90          [PENT,3DNOW]
PFCMPGT mm1,mm2/m64            ; 0F 0F /r A0          [PENT,3DNOW]
```

The `PFCMPxx` instructions compare the packed single-point FP values in the source and destination operands, and set the destination according to the result. If the condition is true, the destination is set to all 1s, otherwise it's set to all 0s.

- `PFCMPEQ` tests whether dst == src;
- `PFCMPGE` tests whether dst >= src;
- `PFCMPGT` tests whether dst > src.

A.5.215 `PFMAX`: Packed Single-Precision FP Maximum

```
PFMAX mm1,mm2/m64              ; 0F 0F /r A4          [PENT,3DNOW]
```

`PFMAX` returns the higher of each pair of single-precision FP values. If the higher value is zero, it is returned as positive zero.

A.5.216 `PFMIN`: Packed Single-Precision FP Minimum

```
PFMIN mm1,mm2/m64             ; 0F 0F /r 94          [PENT,3DNOW]
```

`PFMIN` returns the lower of each pair of single-precision FP values. If the lower value is zero, it is returned as positive zero.

A.5.217 `PFMUL`: Packed Single-Precision FP Multiply

```
PFMUL mm1,mm2/m64             ; 0F 0F /r B4          [PENT,3DNOW]
```

`PFMUL` returns the product of each pair of single-precision FP values.

```
dst[0-31]  := dst[0-31]  * src[0-31],
dst[32-63] := dst[32-63] * src[32-63].
```

A.5.218 `PFNACC`: Packed Single-Precision FP Negative Accumulate

```
PFNACC mm1,mm2/m64            ; 0F 0F /r 8A          [PENT,3DNOW]
```

`PFNACC` performs a negative accumulate of the two single-precision FP values in the source and destination registers. The result of the accumulate from the destination register is stored in the low doubleword of the destination, and the result of the source accumulate is stored in the high doubleword of the destination register.

The operation is:

```
dst[0-31]  := dst[0-31] - dst[32-63],
dst[32-63] := src[0-31] - src[32-63].
```

A.5.219 `PFPNACC`: Packed Single-Precision FP Mixed Accumulate

```
PFPNACC mm1,mm2/m64           ; 0F 0F /r 8E          [PENT,3DNOW]
```

`PFPNACC` performs a positive accumulate of the two single-precision FP values in the source register and a negative accumulate of the destination register. The result of the accumulate from the destination register is stored in the low doubleword of the destination, and the result of the source accumulate is stored in the high doubleword of the destination register.

The operation is:

```
dst[0-31]  := dst[0-31] - dst[32-63],
dst[32-63] := src[0-31] + src[32-63].
```

A.5.220 `PFRCP`: Packed Single-Precision FP Reciprocal Approximation

```
PFRCP mm1,mm2/m64             ; 0F 0F /r 96          [PENT,3DNOW]
```

`PFRCP` performs a low precision estimate of the reciprocal of the low-order single-precision FP value in the source operand, storing the result in both halves of the destination register. The result is accurate to 14 bits.

For higher precision reciprocals, this instruction should be followed by two more instructions: `PFRCPIT1` (section A.5.221) and `PFRCPIT2` (section A.5.221). This will result in a 24-bit accuracy. For more details, see the AMD 3DNow! technology manual.

A.5.221 `PFRCPIT1`: Packed Single-Precision FP Reciprocal, First Iteration Step

```
PFRCPIT1 mm1,mm2/m64          ; 0F 0F /r A6          [PENT,3DNOW]
```

`PFRCPIT1` performs the first intermediate step in the calculation of the reciprocal of a single-precision FP value. The first source value (`mm1` is the original value, and the second source value (`mm2/m64` is the result of a `PFRCP` instruction.

For the final step in a reciprocal, returning the full 24-bit accuracy of a single-precision FP value, see `PFRCPIT2` (section A.5.222). For more details, see the AMD 3DNow! technology manual.

A.5.222 `PFRCPIT2`: Packed Single-Precision FP Reciprocal/ Reciprocal Square Root, Second Iteration Step

```
PFRCPIT2 mm1,mm2/m64          ; 0F 0F /r B6          [PENT,3DNOW]
```

`PFRCPIT2` performs the second and final intermediate step in the calculation of a reciprocal or reciprocal square root, refining the values returned by the `PFRCP` and `PFRSQRT` instructions, respectively.

The first source value (`mm1`) is the output of either a `PFRCPIT1` or a `PFRSQIT1` instruction, and the second source is the output of either the `PFRCP` or the `PFRSQRT` instruction. For more details, see the AMD 3DNow! technology manual.

A.5.223 `PFRSQIT1`: Packed Single-Precision FP Reciprocal Square Root, First Iteration Step

```
PFRSQIT1 mm1,mm2/m64          ; 0F 0F /r A7          [PENT,3DNOW]
```

`PFRSQIT1` performs the first intermediate step in the calculation of the reciprocal square root of a single-precision FP value. The first source value (`mm1` is the square of the result of a `PFRSQRT` instruction, and the second source value (`mm2/m64` is the original value.

For the final step in a calculation, returning the full 24-bit accuracy of a single-precision FP value, see `PFRCPIT2` (section A.5.222). For more details, see the AMD 3DNow! technology manual.

A.5.224 `PFRSQRT`: Packed Single-Precision FP Reciprocal Square Root Approximation

```
PFRSQRT mm1,mm2/m64           ; 0F 0F /r 97          [PENT,3DNOW]
```

`PFRSQRT` performs a low precision estimate of the reciprocal square root of the low-order single-precision FP value in the source operand, storing the result in both halves of the destination register. The result is accurate to 15 bits.

For higher precision reciprocals, this instruction should be followed by two more instructions: `PFRSQIT1` (section A.5.223) and `PFRCPIT2` (section A.5.221). This will result in a 24-bit accuracy. For more details, see the AMD 3DNow! technology manual.

A.5.225 PFSUB: Packed Single-Precision FP Subtract

```
PFSUB mm1,mm2/m64             ; 0F 0F /r 9A           [PENT,3DNOW]
```

PFSUB subtracts the single-precision FP values in the source from those in the destination, and stores the result in the destination operand.

```
dst[0-31]  := dst[0-31]  - src[0-31],
dst[32-63] := dst[32-63] - src[32-63].
```

A.5.226 PFSUBR: Packed Single-Precision FP Reverse Subtract

```
PFSUBR mm1,mm2/m64            ; 0F 0F /r AA           [PENT,3DNOW]
```

PFSUBR subtracts the single-precision FP values in the destination from those in the source, and stores the result in the destination operand.

```
dst[0-31]  := src[0-31]  - dst[0-31],
dst[32-63] := src[32-63] - dst[32-63].
```

A.5.227 PI2FD: Packed Doubleword Integer to Single-Precision FP Convert

```
PI2FD mm1,mm2/m64             ; 0F 0F /r 0D           [PENT,3DNOW]
```

PF2ID converts two signed 32-bit integers in the source operand to single-precision FP values, using truncation of significant digits, and stores them in the destination operand.

A.5.228 PI2FW: Packed Word Integer to Single-Precision FP Convert

```
PI2FW mm1,mm2/m64             ; 0F 0F /r 0C           [PENT,3DNOW]
```

PI2FW converts two signed 16-bit integers in the source operand to single-precision FP values, and stores them in the destination operand. The input values are in the low word of each doubleword.

A.5.229 PINSRW: Insert Word

```
PINSRW mm,r16/r32/m16,imm8    ;0F C4 /r ib      [KATMAI,MMX]
PINSRW xmm,r16/r32/m16,imm8   ;66 0F C4 /r ib   [WILLAMETTE,SSE2]
```

PINSRW loads a word from a 16-bit register (or the low half of a 32-bit register), or from memory, and loads it to the word position in the destination register, pointed at by the count operand (third operand). If the destination is an MMX register, the low two bits of the count byte are used, if it is an XMM register the low 3 bits are used. The insertion is done in such a way that the other words from the destination register are left untouched.

A.5.230 PMACHRIW: Packed Multiply and Accumulate with Rounding

```
PMACHRIW mm,m64               ; 0F 5E /r              [CYRIX,MMX]
```

PMACHRIW takes two packed 16-bit integer inputs, multiplies the values in the inputs, rounds on bit 15 of each result, then adds bits 15-30 of each result to the corresponding position of the *implied* destination register.

The operation of this instruction is:

```
dstI[0-15]  := dstI[0-15]  + (mm[0-15] *m64[0-15]
                                         + 0x00004000)[15-30],
dstI[16-31] := dstI[16-31] + (mm[16-31]*m64[16-31]
                                         + 0x00004000)[15-30],
dstI[32-47] := dstI[32-47] + (mm[32-47]*m64[32-47]
                                         + 0x00004000)[15-30],
dstI[48-63] := dstI[48-63] + (mm[48-63]*m64[48-63]
                                         + 0x00004000)[15-30].
```

Note that PMACHRIW cannot take a register as its second source operand.

A.5.231 PMADDWD: MMX Packed Multiply and Add

```
PMADDWD mm1,mm2/m64              ; 0F F5 /r             [PENT,MMX]
PMADDWD xmm1,xmm2/m128           ; 66 0F F5 /r     [WILLAMETTE,SSE2]
```

PMADDWD treats its two inputs as vectors of signed words. It multiplies corresponding elements of the two operands, giving doubleword results. These are then added together in pairs and stored in the destination operand.

The operation of this instruction is:

```
dst[0-31]   := (dst[0-15] * src[0-15])
                            + (dst[16-31] * src[16-31]);
dst[32-63]  := (dst[32-47] * src[32-47])
                            + (dst[48-63] * src[48-63]);
```

The following apply to the SSE version of the instruction:

```
dst[64-95]  := (dst[64-79] * src[64-79])
                            + (dst[80-95] * src[80-95]);
dst[96-127] := (dst[96-111] * src[96-111])
                            + (dst[112-127] * src[112-127]).
```

A.5.232 PMAGW: MMX Packed Magnitude

```
PMAGW mm1,mm2/m64                ; 0F 52 /r             [CYRIX,MMX]
```

PMAGW, specific to the Cyrix MMX extensions, treats both its operands as vectors of four signed words. It compares the absolute values of the words in corresponding positions, and sets each word of the destination (first) operand to whichever of the two words in that position had the larger absolute value.

A.5.233 PMAXSW: Packed Signed Integer Word Maximum

```
PMAXSW mm1,mm2/m64               ; 0F EE /r        [KATMAI,MMX]
PMAXSW xmm1,xmm2/m128            ; 66 0F EE /r     [WILLAMETTE,SSE2]
```

PMAXSW compares each pair of words in the two source operands, and for each pair it stores the maximum value in the destination register.

A.5.234 PMAXUB: Packed Unsigned Integer Byte Maximum

```
PMAXUB mm1,mm2/m64              ; 0F DE /r         [KATMAI,MMX]
PMAXUB xmm1,xmm2/m128           ; 66 0F DE /r      [WILLAMETTE,SSE2]
```

PMAXUB compares each pair of bytes in the two source operands, and for each pair it stores the maximum value in the destination register.

A.5.235 PMINSW: Packed Signed Integer Word Minimum

```
PMINSW mm1,mm2/m64              ; 0F EA /r         [KATMAI,MMX]
PMINSW xmm1,xmm2/m128           ; 66 0F EA /r      [WILLAMETTE,SSE2]
```

PMINSW compares each pair of words in the two source operands, and for each pair it stores the minimum value in the destination register.

A.5.236 PMINUB: Packed Unsigned Integer Byte Minimum

```
PMINUB mm1,mm2/m64              ; 0F DA /r         [KATMAI,MMX]
PMINUB xmm1,xmm2/m128           ; 66 0F DA /r      [WILLAMETTE,SSE2]
```

PMINUB compares each pair of bytes in the two source operands, and for each pair it stores the minimum value in the destination register.

A.5.237 PMOVMSKB: Move Byte Mask To Integer

```
PMOVMSKB reg32,mm               ; 0F D7 /r         [KATMAI,MMX]
PMOVMSKB reg32,xmm              ; 66 0F D7 /r      [WILLAMETTE,SSE2]
```

PMOVMSKB returns an 8-bit or 16-bit mask formed of the most significant bits of each byte of source operand (8-bits for an MMX register, 16-bits for an XMM register).

A.5.238 PMULHRWC, PMULHRIW: Multiply Packed 16-bit Integers With Rounding, and Store High Word

```
PMULHRWC mm1,mm2/m64           ; 0F 59 /r                   [CYRIX,MMX]
PMULHRIW mm1,mm2/m64           ; 0F 5D /r                   [CYRIX,MMX]
```

These instructions take two packed 16-bit integer inputs, multiply the values in the inputs, round on bit 15 of each result, then store bits 15-30 of each result to the corresponding position of the destination register.

- For PMULHRWC, the destination is the first source operand.
- For PMULHRIW, the destination is an implied register (worked out as described for PADDSIW (section A.5.200)).

The operation of this instruction is:

```
dst[0-15]  := (src1[0-15] *src2[0-15]  + 0x00004000)[15-30]
dst[16-31] := (src1[16-31]*src2[16-31] + 0x00004000)[15-30]
dst[32-47] := (src1[32-47]*src2[32-47] + 0x00004000)[15-30]
dst[48-63] := (src1[48-63]*src2[48-63] + 0x00004000)[15-30]
```

See also PMULHRWA (section A.5.239) for a 3DNow! version of this instruction.

A.5.239 `PMULHRWA`: Multiply Packed 16-bit Integers With Rounding, and Store High Word

```
PMULHRWA mm1,mm2/m64          ; 0F 0F /r B7     [PENT,3DNOW]
```

`PMULHRWA` takes two packed 16-bit integer inputs, multiplies the values in the inputs, rounds on bit 16 of each result, then stores bits 16-31 of each result to the corresponding position of the destination register.

The operation of this instruction is:

```
dst[0-15]  := (src1[0-15] *src2[0-15]  + 0x00008000)[16-31];
dst[16-31] := (src1[16-31]*src2[16-31] + 0x00008000)[16-31];
dst[32-47] := (src1[32-47]*src2[32-47] + 0x00008000)[16-31];
dst[48-63] := (src1[48-63]*src2[48-63] + 0x00008000)[16-31].
```

See also `PMULHRWC` (section A.5.238) for a Cyrix version of this instruction.

A.5.240 `PMULHUW`: Multiply Packed 16-bit Integers, and Store High Word

```
PMULHUW mm1,mm2/m64           ; 0F E4 /r        [KATMAI,MMX]
PMULHUW xmm1,xmm2/m128        ; 66 0F E4 /r     [WILLAMETTE,SSE2]
```

`PMULHUW` takes two packed unsigned 16-bit integer inputs, multiplies the values in the inputs, then stores bits 16-31 of each result to the corresponding position of the destination register.

A.5.241 `PMULHW`, `PMULLW`: Multiply Packed 16-bit Integers, and Store

```
PMULHW mm1,mm2/m64            ; 0F E5 /r             [PENT,MMX]
PMULLW mm1,mm2/m64            ; 0F D5 /r             [PENT,MMX]

PMULHW xmm1,xmm2/m128         ; 66 0F E5 /r     [WILLAMETTE,SSE2]
PMULLW xmm1,xmm2/m128         ; 66 0F D5 /r     [WILLAMETTE,SSE2]
```

`PMULxW` takes two packed unsigned 16-bit integer inputs, and multiplies the values in the inputs, forming doubleword results.

- `PMULHW` then stores the top 16 bits of each doubleword in the destination (first) operand;
- `PMULLW` stores the bottom 16 bits of each doubleword in the destination operand.

A.5.242 `PMULUDQ`: Multiply Packed Unsigned 32-bit Integers, and Store.

```
PMULUDQ mm1,mm2/m64           ; 0F F4 /r        [WILLAMETTE,SSE2]
PMULUDQ xmm1,xmm2/m128        ; 66 0F F4 /r     [WILLAMETTE,SSE2]
```

`PMULUDQ` takes two packed unsigned 32-bit integer inputs, and multiplies the values in the inputs, forming quadword results. The source is either an unsigned doubleword in the low doubleword of a 64-bit operand, or it's two unsigned doublewords in the first and third doublewords of a 128-bit operand. This produces either one or two 64-bit results, which are stored in the respective quadword locations of the destination register.

The operation is:

```
dst[0-63]   := dst[0-31]  * src[0-31];
dst[64-127] := dst[64-95] * src[64-95].
```

A.5.243 `PMVccZB`: MMX Packed Conditional Move

```
PMVZB mmxreg,mem64            ; 0F 58 /r             [CYRIX,MMX]
PMVNZB mmxreg,mem64           ; 0F 5A /r             [CYRIX,MMX]
PMVLZB mmxreg,mem64           ; 0F 5B /r             [CYRIX,MMX]
PMVGEZB mmxreg,mem64          ; 0F 5C /r             [CYRIX,MMX]
```

These instructions, specific to the Cyrix MMX extensions, perform parallel conditional moves. The two input operands are treated as vectors of eight bytes. Each byte of the destination (first) operand is either written from the corresponding byte of the source (second) operand, or left alone, depending on the value of the byte in the *implied* operand (specified in the same way as `PADDSIW`, in section A.5.200).

- `PMVZB` performs each move if the corresponding byte in the implied operand is zero;
- `PMVNZB` moves if the byte is non-zero;
- `PMVLZB` moves if the byte is less than zero;
- `PMVGEZB` moves if the byte is greater than or equal to zero.

Note that these instructions cannot take a register as their second source operand.

A.5.244 `POP`: Pop Data from Stack

```
POP reg16                     ; o16 58+r             [8086]
POP reg32                     ; o32 58+r             [386]

POP r/m16                     ; o16 8F /0            [8086]
POP r/m32                     ; o32 8F /0            [386]

POP CS                        ; 0F                   [8086,UNDOC]
POP DS                        ; 1F                   [8086]
POP ES                        ; 07                   [8086]
POP SS                        ; 17                   [8086]
POP FS                        ; 0F A1                [386]
POP GS                        ; 0F A9                [386]
```

`POP` loads a value from the stack (from `[SS:SP]` or `[SS:ESP]`) and then increments the stack pointer.

The address-size attribute of the instruction determines whether `SP` or `ESP` is used as the stack pointer: to deliberately override the default given by the `BITS` setting, you can use an `a16` or `a32` prefix.

The operand-size attribute of the instruction determines whether the stack pointer is incremented by 2 or 4: this means that segment register pops in `BITS 32` mode will pop 4 bytes off the stack and discard the upper two of them. If you need to override that, you can use an `o16` or `o32` prefix.

The above opcode listings give two forms for general-purpose register pop instructions: for example, `POP BX` has the two forms `5B` and `8F C3`. NASM will always generate the shorter form when given `POP BX`. NDISASM will disassemble both.

`POP CS` is not a documented instruction, and is not supported on any processor above the 8086 (since they use `0Fh` as an opcode prefix for instruction set extensions). However, at least some 8086 processors do support it, and so NASM generates it for completeness. Current versions of NASM silently accept this instruction.

A.5.245 POPAx: Pop All General-Purpose Registers

```
POPA                          ; 61                     [186]
POPAW                         ; o16 61                 [186]
POPAD                         ; o32 61                 [386]
```

- `POPAW` pops a word from the stack into each of, successively, `DI`, `SI`, `BP`, nothing (it discards a word from the stack which was a placeholder for `SP`), `BX`, `DX`, `CX` and `AX`. It is intended to reverse the operation of `PUSHAW` (see section A.5.264), but it ignores the value for `SP` that was pushed on the stack by `PUSHAW`.
- `POPAD` pops twice as much data, and places the results in `EDI`, `ESI`, `EBP`, nothing (placeholder for `ESP`), `EBX`, `EDX`, `ECX` and `EAX`. It reverses the operation of `PUSHAD`.

`POPA` is an alias mnemonic for either `POPAW` or `POPAD`, depending on the current `BITS` setting.

Note that the registers are popped in reverse order of their numeric values in opcodes (see section A.2.1).

A.5.246 POPFx: Pop Flags Register

```
POPF                          ; 9D                     [8086]
POPFW                         ; o16 9D                 [8086]
POPFD                         ; o32 9D                 [386]
```

- `POPFW` pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386).
- `POPFD` pops a doubleword and stores it in the entire flags register.

`POPF` is an alias mnemonic for either `POPFW` or `POPFD`, depending on the current `BITS` setting.

See also `PUSHF` (section A.5.265).

A.5.247 POR: MMX Bitwise OR

```
POR mm1,mm2/m64               ; 0F EB /r               [PENT,MMX]
POR xmm1,xmm2/m128            ; 66 0F EB /r     [WILLAMETTE,SSE2]
```

`POR` performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

A.5.248 PREFETCH: Prefetch Data Into Caches

```
PREFETCH mem8                 ; 0F 0D /0               [PENT,3DNOW]
PREFETCHW mem8                ; 0F 0D /1               [PENT,3DNOW]
```

`PREFETCH` and `PREFETCHW` fetch the line of data from memory that contains the specified byte. `PREFETCHW` performs differently on the Athlon to earlier processors.

For more details, see the 3DNow! Technology Manual.

A.5.249 PREFETCHh: Prefetch Data Into Caches

```
PREFETCHNTA m8                ; 0F 18 /0        [KATMAI]
PREFETCHT0 m8                 ; 0F 18 /1        [KATMAI]
PREFETCHT1 m8                 ; 0F 18 /2        [KATMAI]
PREFETCHT2 m8                 ; 0F 18 /3        [KATMAI]
```

The `PREFETCHh` instructions fetch the line of data from memory that contains the specified byte. It is placed in the cache according to rules specified by locality hints `h`:

The hints are:

- `T0` (temporal data) - prefetch data into all levels of the cache hierarchy.
- `T1` (temporal data with respect to first level cache) - prefetch data into level 2 cache and higher.
- `T2` (temporal data with respect to second level cache) - prefetch data into level 2 cache and higher.
- `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.

Note that this group of instructions doesn't provide a guarantee that the data will be in the cache when it is needed. For more details, see the Intel IA32 Software Developer Manual, Volume 2.

A.5.250 PSADBW: Packed Sum of Absolute Differences

```
PSADBW mm1,mm2/m64            ; 0F F6 /r        [KATMAI,MMX]
PSADBW xmm1,xmm2/m128         ; 66 0F F6 /r     [WILLAMETTE,SSE2]
```

`PSADBW` The PSADBW instruction computes the absolute value of the difference of the packed unsigned bytes in the two source operands. These differences are then summed to produce a word result in the lower 16-bit field of the destination register; the rest of the register is cleared. The destination operand is an `MMX` or an `XMM` register. The source operand can either be a register or a memory operand.

A.5.251 PSHUFD: Shuffle Packed Doublewords

```
PSHUFD xmm1,xmm2/m128,imm8    ; 66 0F 70 /r ib  [WILLAMETTE,SSE2]
```

`PSHUFD` shuffles the doublewords in the source (second) operand according to the encoding specified by imm8, and stores the result in the destination (first) operand.

Bits 0 and 1 of imm8 encode the source position of the doubleword to be copied to position 0 in the destination operand. Bits 2 and 3 encode for position 1, bits 4 and 5 encode for position 2, and bits 6 and 7 encode for position 3. For example, an encoding of 10 in bits 0 and 1 of imm8 indicates that the doubleword at bits 64-95 of the source operand will be copied to bits 0-31 of the destination.

A.5.252 PSHUFHW: Shuffle Packed High Words

```
PSHUFHW xmm1,xmm2/m128,imm8   ; F3 0F 70 /r ib  [WILLAMETTE,SSE2]
```

`PSHUFW` shuffles the words in the high quadword of the source (second) operand according to the encoding specified by imm8, and stores the result in the high quadword of the destination (first) operand.

The operation of this instruction is similar to the `PSHUFW` instruction, except that the source and destination are the top quadword of a 128-bit operand, instead of being 64-bit operands. The low quadword is copied from the source to the destination without any changes.

A.5.253 `PSHUFLW`: Shuffle Packed Low Words

```
PSHUFLW xmm1,xmm2/m128,imm8   ; F2 0F 70 /r ib  [WILLAMETTE,SSE2]
```

`PSHUFLW` shuffles the words in the low quadword of the source (second) operand according to the encoding specified by imm8, and stores the result in the low quadword of the destination (first) operand.

The operation of this instruction is similar to the `PSHUFW` instruction, except that the source and destination are the low quadword of a 128-bit operand, instead of being 64-bit operands. The high quadword is copied from the source to the destination without any changes.

A.5.254 `PSHUFW`: Shuffle Packed Words

```
PSHUFW mm1,mm2/m64,imm8       ; 0F 70 /r ib     [KATMAI,MMX]
```

`PSHUFW` shuffles the words in the source (second) operand according to the encoding specified by imm8, and stores the result in the destination (first) operand.

Bits 0 and 1 of imm8 encode the source position of the word to be copied to position 0 in the destination operand. Bits 2 and 3 encode for position 1, bits 4 and 5 encode for position 2, and bits 6 and 7 encode for position 3. For example, an encoding of 10 in bits 0 and 1 of imm8 indicates that the word at bits 32-47 of the source operand will be copied to bits 0-15 of the destination.

A.5.255 `PSLLx`: Packed Data Bit Shift Left Logical

```
PSLLW mm1,mm2/m64             ; 0F F1 /r             [PENT,MMX]
PSLLW mm,imm8                 ; 0F 71 /6 ib          [PENT,MMX]

PSLLW xmm1,xmm2/m128          ; 66 0F F1 /r     [WILLAMETTE,SSE2]
PSLLW xmm,imm8                ; 66 0F 71 /6 ib  [WILLAMETTE,SSE2]

PSLLD mm1,mm2/m64             ; 0F F2 /r             [PENT,MMX]
PSLLD mm,imm8                 ; 0F 72 /6 ib          [PENT,MMX]

PSLLD xmm1,xmm2/m128          ; 66 0F F2 /r     [WILLAMETTE,SSE2]
PSLLD xmm,imm8                ; 66 0F 72 /6 ib  [WILLAMETTE,SSE2]

PSLLQ mm1,mm2/m64             ; 0F F3 /r             [PENT,MMX]
PSLLQ mm,imm8                 ; 0F 73 /6 ib          [PENT,MMX]

PSLLQ xmm1,xmm2/m128          ; 66 0F F3 /r     [WILLAMETTE,SSE2]
PSLLQ xmm,imm8                ; 66 0F 73 /6 ib  [WILLAMETTE,SSE2]

PSLLDQ xmm1,imm8              ; 66 0F 73 /7 ib  [WILLAMETTE,SSE2]
```

`PSLLx` performs logical left shifts of the data elements in the destination (first) operand, moving each bit in the separate elements left by the number of bits specified in the source (second) operand, clearing the low-order bits as they are vacated. PSLLDQ shifts bytes, not bits.

- PSLLW shifts word sized elements.
- PSLLD shifts doubleword sized elements.
- PSLLQ shifts quadword sized elements.
- PSLLDQ shifts double quadword sized elements.

A.5.256 PSRAx: Packed Data Bit Shift Right Arithmetic

```
PSRAW mm1,mm2/m64             ; 0F E1 /r             [PENT,MMX]
PSRAW mm,imm8                 ; 0F 71 /4 ib          [PENT,MMX]

PSRAW xmm1,xmm2/m128          ; 66 0F E1 /r     [WILLAMETTE,SSE2]
PSRAW xmm,imm8                ; 66 0F 71 /4 ib  [WILLAMETTE,SSE2]

PSRAD mm1,mm2/m64             ; 0F E2 /r             [PENT,MMX]
PSRAD mm,imm8                 ; 0F 72 /4 ib          [PENT,MMX]

PSRAD xmm1,xmm2/m128          ; 66 0F E2 /r     [WILLAMETTE,SSE2]
PSRAD xmm,imm8                ; 66 0F 72 /4 ib  [WILLAMETTE,SSE2]
```

PSRAx performs arithmetic right shifts of the data elements in the destination (first) operand, moving each bit in the separate elements right by the number of bits specified in the source (second) operand, setting the high-order bits to the value of the original sign bit.

- PSRAW shifts word sized elements.
- PSRAD shifts doubleword sized elements.

A.5.257 PSRLx: Packed Data Bit Shift Right Logical

```
PSRLW mm1,mm2/m64             ; 0F D1 /r             [PENT,MMX]
PSRLW mm,imm8                 ; 0F 71 /2 ib          [PENT,MMX]

PSRLW xmm1,xmm2/m128          ; 66 0F D1 /r     [WILLAMETTE,SSE2]
PSRLW xmm,imm8                ; 66 0F 71 /2 ib  [WILLAMETTE,SSE2]

PSRLD mm1,mm2/m64             ; 0F D2 /r             [PENT,MMX]
PSRLD mm,imm8                 ; 0F 72 /2 ib          [PENT,MMX]

PSRLD xmm1,xmm2/m128          ; 66 0F D2 /r     [WILLAMETTE,SSE2]
PSRLD xmm,imm8                ; 66 0F 72 /2 ib  [WILLAMETTE,SSE2]

PSRLQ mm1,mm2/m64             ; 0F D3 /r             [PENT,MMX]
PSRLQ mm,imm8                 ; 0F 73 /2 ib          [PENT,MMX]

PSRLQ xmm1,xmm2/m128          ; 66 0F D3 /r     [WILLAMETTE,SSE2]
PSRLQ xmm,imm8                ; 66 0F 73 /2 ib  [WILLAMETTE,SSE2]

PSRLDQ xmm1,imm8              ; 66 0F 73 /3 ib  [WILLAMETTE,SSE2]
```

PSRLx performs logical right shifts of the data elements in the destination (first) operand, moving each bit in the separate elements right by the number of bits specified in the source (second) operand, clearing the high-order bits as they are vacated. PSRLDQ shifts bytes, not bits.

- PSRLW shifts word sized elements.
- PSRLD shifts doubleword sized elements.
- PSRLQ shifts quadword sized elements.
- PSRLDQ shifts double quadword sized elements.

A.5.258 PSUBx: Subtract Packed Integers

```
PSUBB mm1,mm2/m64             ; 0F F8 /r              [PENT,MMX]
PSUBW mm1,mm2/m64             ; 0F F9 /r              [PENT,MMX]
PSUBD mm1,mm2/m64             ; 0F FA /r              [PENT,MMX]
PSUBQ mm1,mm2/m64             ; 0F FB /r        [WILLAMETTE,SSE2]

PSUBB xmm1,xmm2/m128          ; 66 0F F8 /r     [WILLAMETTE,SSE2]
PSUBW xmm1,xmm2/m128          ; 66 0F F9 /r     [WILLAMETTE,SSE2]
PSUBD xmm1,xmm2/m128          ; 66 0F FA /r     [WILLAMETTE,SSE2]
PSUBQ xmm1,xmm2/m128          ; 66 0F FB /r     [WILLAMETTE,SSE2]
```

PSUBx subtracts packed integers in the source operand from those in the destination operand. It doesn't differentiate between signed and unsigned integers, and doesn't set any of the flags.

- PSUBB operates on byte sized elements.
- PSUBW operates on word sized elements.
- PSUBD operates on doubleword sized elements.
- PSUBQ operates on quadword sized elements.

A.5.259 PSUBSxx, PSUBUSx: Subtract Packed Integers With Saturation

```
PSUBSB mm1,mm2/m64            ; 0F E8 /r              [PENT,MMX]
PSUBSW mm1,mm2/m64            ; 0F E9 /r              [PENT,MMX]

PSUBSB xmm1,xmm2/m128         ; 66 0F E8 /r     [WILLAMETTE,SSE2]
PSUBSW xmm1,xmm2/m128         ; 66 0F E9 /r     [WILLAMETTE,SSE2]

PSUBUSB mm1,mm2/m64           ; 0F D8 /r              [PENT,MMX]
PSUBUSW mm1,mm2/m64           ; 0F D9 /r              [PENT,MMX]

PSUBUSB xmm1,xmm2/m128        ; 66 0F D8 /r     [WILLAMETTE,SSE2]
PSUBUSW xmm1,xmm2/m128        ; 66 0F D9 /r     [WILLAMETTE,SSE2]
```

PSUBSx and PSUBUSx subtracts packed integers in the source operand from those in the destination operand, and use saturation for results that are outside the range supported by the destination operand.

- PSUBSB operates on signed bytes, and uses signed saturation on the results.
- PSUBSW operates on signed words, and uses signed saturation on the results.

- PSUBUSB operates on unsigned bytes, and uses signed saturation on the results.
- PSUBUSW operates on unsigned words, and uses signed saturation on the results.

A.5.260 PSUBSIW: MMX Packed Subtract with Saturation to Implied Destination

```
PSUBSIW mm1,mm2/m64             ; 0F 55 /r             [CYRIX,MMX]
```

PSUBSIW, specific to the Cyrix extensions to the MMX instruction set, performs the same function as PSUBSW, except that the result is not placed in the register specified by the first operand, but instead in the implied destination register, specified as for PADDSIW (section A.5.200).

A.5.261 PSWAPD: Swap Packed Data

```
PSWAPD mm1,mm2/m64              ; 0F 0F /r BB     [PENT,3DNOW]
```

PSWAPD swaps the packed doublewords in the source operand, and stores the result in the destination operand.

In the K6-2 and K6-III processors, this opcode uses the mnemonic PSWAPW, and it swaps the order of words when copying from the source to the destination.

The operation in the K6-2 and K6-III processors is

```
dst[0-15]  = src[48-63];
dst[16-31] = src[32-47];
dst[32-47] = src[16-31];
dst[48-63] = src[0-15].
```

The operation in the K6-x+, ATHLON and later processors is:

```
dst[0-31]  = src[32-63];
dst[32-63] = src[0-31].
```

A.5.262 PUNPCKxxx: Unpack and Interleave Data

```
PUNPCKHBW mm1,mm2/m64           ; 0F 68 /r             [PENT,MMX]
PUNPCKHWD mm1,mm2/m64           ; 0F 69 /r             [PENT,MMX]
PUNPCKHDQ mm1,mm2/m64           ; 0F 6A /r             [PENT,MMX]

PUNPCKHBW xmm1,xmm2/m128        ; 66 0F 68 /r     [WILLAMETTE,SSE2]
PUNPCKHWD xmm1,xmm2/m128        ; 66 0F 69 /r     [WILLAMETTE,SSE2]
PUNPCKHDQ xmm1,xmm2/m128        ; 66 0F 6A /r     [WILLAMETTE,SSE2]
PUNPCKHQDQ xmm1,xmm2/m128       ; 66 0F 6D /r     [WILLAMETTE,SSE2]

PUNPCKLBW mm1,mm2/m32           ; 0F 60 /r             [PENT,MMX]
PUNPCKLWD mm1,mm2/m32           ; 0F 61 /r             [PENT,MMX]
PUNPCKLDQ mm1,mm2/m32           ; 0F 62 /r             [PENT,MMX]

PUNPCKLBW xmm1,xmm2/m128        ; 66 0F 60 /r     [WILLAMETTE,SSE2]
PUNPCKLWD xmm1,xmm2/m128        ; 66 0F 61 /r     [WILLAMETTE,SSE2]
```

```
PUNPCKLDQ xmm1,xmm2/m128      ; 66 0F 62 /r     [WILLAMETTE,SSE2]
PUNPCKLQDQ xmm1,xmm2/m128     ; 66 0F 6C /r     [WILLAMETTE,SSE2]
```

`PUNPCKxx` all treat their operands as vectors, and produce a new vector generated by interleaving elements from the two inputs. The `PUNPCKHxx` instructions start by throwing away the bottom half of each input operand, and the `PUNPCKLxx` instructions throw away the top half.

The remaining elements, are then interleaved into the destination, alternating elements from the second (source) operand and the first (destination) operand: so the leftmost part of each element in the result always comes from the second operand, and the rightmost from the destination.

- `PUNPCKxBW` works a byte at a time, producing word sized output elements.
- `PUNPCKxWD` works a word at a time, producing doubleword sized output elements.
- `PUNPCKxDQ` works a doubleword at a time, producing quadword sized output elements.
- `PUNPCKxQDQ` works a quadword at a time, producing double quadword sized output elements.

So, for example, for `MMX` operands, if the first operand held `0x7A6A5A4A3A2A1A0A` and the second held `0x7B6B5B4B3B2B1B0B`, then:

- `PUNPCKHBW` would return `0x7B7A6B6A5B5A4B4A`.
- `PUNPCKHWD` would return `0x7B6B7A6A5B4B5A4A`.
- `PUNPCKHDQ` would return `0x7B6B5B4B7A6A5A4A`.
- `PUNPCKLBW` would return `0x3B3A2B2A1B1A0B0A`.
- `PUNPCKLWD` would return `0x3B2B3A2A1B0B1A0A`.
- `PUNPCKLDQ` would return `0x3B2B1B0B3A2A1A0A`.

A.5.263 `PUSH`: Push Data on Stack

```
PUSH reg16                    ; o16 50+r              [8086]
PUSH reg32                    ; o32 50+r              [386]

PUSH r/m16                    ; o16 FF /6             [8086]
PUSH r/m32                    ; o32 FF /6             [386]

PUSH CS                       ; 0E                    [8086]
PUSH DS                       ; 1E                    [8086]
PUSH ES                       ; 06                    [8086]
PUSH SS                       ; 16                    [8086]
PUSH FS                       ; 0F A0                 [386]
PUSH GS                       ; 0F A8                 [386]

PUSH imm8                     ; 6A ib                 [186]
PUSH imm16                    ; o16 68 iw             [186]
PUSH imm32                    ; o32 68 id             [386]
```

PUSH decrements the stack pointer (SP or ESP) by 2 or 4, and then stores the given value at [SS:SP] or [SS:ESP].

The address-size attribute of the instruction determines whether SP or ESP is used as the stack pointer: to deliberately override the default given by the BITS setting, you can use an a16 or a32 prefix.

The operand-size attribute of the instruction determines whether the stack pointer is decremented by 2 or 4: this means that segment register pushes in BITS 32 mode will push 4 bytes on the stack, of which the upper two are undefined. If you need to override that, you can use an o16 or o32 prefix.

The above opcode listings give two forms for general-purpose register push instructions: for example, PUSH BX has the two forms 53 and FF F3. NASM will always generate the shorter form when given PUSH BX. NDISASM will disassemble both.

Unlike the undocumented and barely supported POP CS, PUSH CS is a perfectly valid and sensible instruction, supported on all processors.

The instruction PUSH SP may be used to distinguish an 8086 from later processors: on an 8086, the value of SP stored is the value it has *after* the push instruction, whereas on later processors it is the value *before* the push instruction.

A.5.264 PUSHAx: Push All General-Purpose Registers

```
PUSHA                          ; 60                    [186]
PUSHAD                         ; o32 60                [386]
PUSHAW                         ; o16 60                [186]
```

PUSHAW pushes, in succession, AX, CX, DX, BX, SP, BP, SI and DI on the stack, decrementing the stack pointer by a total of 16.

PUSHAD pushes, in succession, EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI on the stack, decrementing the stack pointer by a total of 32.

In both cases, the value of SP or ESP pushed is its *original* value, as it had before the instruction was executed.

PUSHA is an alias mnemonic for either PUSHAW or PUSHAD, depending on the current BITS setting.

Note that the registers are pushed in order of their numeric values in opcodes (see section A.2.1).

See also POPA (section A.5.245).

A.5.265 PUSHFx: Push Flags Register

```
PUSHF                          ; 9C                    [8086]
PUSHFD                         ; o32 9C                [386]
PUSHFW                         ; o16 9C                [8086]
```

- PUSHFW pushes the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386) onto the stack.
- PUSHFD pushes the entire flags register onto the stack.

PUSHF is an alias mnemonic for either PUSHFW or PUSHFD, depending on the current BITS setting.

See also POPF (section A.5.246).

A.5.266 PXOR: MMX Bitwise XOR

```
PXOR mm1,mm2/m64              ; 0F EF /r                [PENT,MMX]
PXOR xmm1,xmm2/m128           ; 66 0F EF /r     [WILLAMETTE,SSE2]
```

PXOR performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

A.5.267 RCL, RCR: Bitwise Rotate through Carry Bit

```
RCL r/m8,1                    ; D0 /2                   [8086]
RCL r/m8,CL                   ; D2 /2                   [8086]
RCL r/m8,imm8                 ; C0 /2 ib                [186]
RCL r/m16,1                   ; o16 D1 /2               [8086]
RCL r/m16,CL                  ; o16 D3 /2               [8086]
RCL r/m16,imm8                ; o16 C1 /2 ib            [186]
RCL r/m32,1                   ; o32 D1 /2               [386]
RCL r/m32,CL                  ; o32 D3 /2               [386]
RCL r/m32,imm8                ; o32 C1 /2 ib            [386]

RCR r/m8,1                    ; D0 /3                   [8086]
RCR r/m8,CL                   ; D2 /3                   [8086]
RCR r/m8,imm8                 ; C0 /3 ib                [186]
RCR r/m16,1                   ; o16 D1 /3               [8086]
RCR r/m16,CL                  ; o16 D3 /3               [8086]
RCR r/m16,imm8                ; o16 C1 /3 ib            [186]
RCR r/m32,1                   ; o32 D1 /3               [386]
RCR r/m32,CL                  ; o32 D3 /3               [386]
RCR r/m32,imm8                ; o32 C1 /3 ib            [386]
```

RCL and RCR perform a 9-bit, 17-bit or 33-bit bitwise rotation operation, involving the given source/destination (first) operand and the carry bit. Thus, for example, in the operation RCL AL,1, a 9-bit rotation is performed in which AL is shifted left by 1, the top bit of AL moves into the carry flag, and the original value of the carry flag is placed in the low bit of AL.

The number of bits to rotate by is given by the second operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.

You can force the longer (186 and upwards, beginning with a C1 or C0 byte) form of RCL foo,1 by using a BYTE prefix: RCL foo,BYTE 1. Similarly with RCR.

A.5.268 RCPPS: Packed Single-Precision FP Reciprocal

```
RCPPS xmm1,xmm2/m128          ; 0F 53 /r        [KATMAI,SSE]
```

RCPPS returns an approximation of the reciprocal of the packed single-precision FP values from xmm2/m128. The maximum error for this approximation is: |Error| <= 1.5 x 2^-12

A.5.269 RCPSS: Scalar Single-Precision FP Reciprocal

```
RCPSS xmm1,xmm2/m128          ; F3 0F 53 /r     [KATMAI,SSE]
```

`RCPSS` returns an approximation of the reciprocal of the lower single-precision FP value from xmm2/m32; the upper three fields are passed through from xmm1. The maximum error for this approximation is: |Error| <= 1.5 x 2^-12

A.5.270 RDMSR: Read Model-Specific Registers

```
RDMSR                         ; 0F 32                [PENT,PRIV]
```

`RDMSR` reads the processor Model-Specific Register (MSR) whose index is stored in `ECX`, and stores the result in `EDX:EAX`. See also `WRMSR` (section A.5.329).

A.5.271 RDPMC: Read Performance-Monitoring Counters

```
RDPMC                         ; 0F 33                [P6]
```

`RDPMC` reads the processor performance-monitoring counter whose index is stored in `ECX`, and stores the result in `EDX:EAX`.

This instruction is available on P6 and later processors and on MMX class processors.

A.5.272 RDSHR: Read SMM Header Pointer Register

```
RDSHR r/m32                   ; 0F 36 /0        [386,CYRIX,SMM]
```

`RDSHR` reads the contents of the SMM header pointer register and saves it to the destination operand, which can be either a 32 bit memory location or a 32 bit register.

See also `WRSHR` (section A.5.330).

A.5.273 RDTSC: Read Time-Stamp Counter

```
RDTSC                         ; 0F 31                [PENT]
```

`RDTSC` reads the processor's time-stamp counter into `EDX:EAX`.

A.5.274 RET, RETF, RETN: Return from Procedure Call

```
RET                           ; C3                   [8086]
RET imm16                     ; C2 iw                [8086]

RETF                          ; CB                   [8086]
RETF imm16                    ; CA iw                [8086]

RETN                          ; C3                   [8086]
RETN imm16                    ; C2 iw                [8086]
```

- `RET`, and its exact synonym `RETN`, pop `IP` or `EIP` from the stack and transfer control to the new address. Optionally, if a numeric second operand is provided, they increment the stack pointer by a further `imm16` bytes after popping the return address.
- `RETF` executes a far return: after popping `IP`/`EIP`, it then pops `CS`, and *then* increments the stack pointer by the optional argument if present.

A.5.275 `ROL`, `ROR`: Bitwise Rotate

```
ROL r/m8,1                    ; D0 /0                [8086]
ROL r/m8,CL                   ; D2 /0                [8086]
ROL r/m8,imm8                 ; C0 /0 ib             [186]
ROL r/m16,1                   ; o16 D1 /0            [8086]
ROL r/m16,CL                  ; o16 D3 /0            [8086]
ROL r/m16,imm8                ; o16 C1 /0 ib         [186]
ROL r/m32,1                   ; o32 D1 /0            [386]
ROL r/m32,CL                  ; o32 D3 /0            [386]
ROL r/m32,imm8                ; o32 C1 /0 ib         [386]

ROR r/m8,1                    ; D0 /1                [8086]
ROR r/m8,CL                   ; D2 /1                [8086]
ROR r/m8,imm8                 ; C0 /1 ib             [186]
ROR r/m16,1                   ; o16 D1 /1            [8086]
ROR r/m16,CL                  ; o16 D3 /1            [8086]
ROR r/m16,imm8                ; o16 C1 /1 ib         [186]
ROR r/m32,1                   ; o32 D1 /1            [386]
ROR r/m32,CL                  ; o32 D3 /1            [386]
ROR r/m32,imm8                ; o32 C1 /1 ib         [386]
```

`ROL` and `ROR` perform a bitwise rotation operation on the given source/destination (first) operand. Thus, for example, in the operation `ROL AL,1`, an 8-bit rotation is performed in which `AL` is shifted left by 1 and the original top bit of `AL` moves round into the low bit.

The number of bits to rotate by is given by the second operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.

You can force the longer (186 and upwards, beginning with a `C1` or `C0` byte) form of `ROL foo,1` by using a `BYTE` prefix: `ROL foo,BYTE 1`. Similarly with `ROR`.

A.5.276 `RSDC`: Restore Segment Register and Descriptor

```
RSDC segreg,m80               ; 0F 79 /r        [486,CYRIX,SMM]
```

`RSDC` restores a segment register (DS, ES, FS, GS, or SS) from mem80, and sets up its descriptor.

A.5.277 `RSLDT`: Restore Segment Register and Descriptor

```
RSLDT m80                     ; 0F 7B /0        [486,CYRIX,SMM]
```

`RSLDT` restores the Local Descriptor Table (LDTR) from mem80.

A.5.278 `RSM`: Resume from System-Management Mode

```
RSM                           ; 0F AA                [PENT]
```

`RSM` returns the processor to its normal operating mode when it was in System-Management Mode.

A.5.279 `RSQRTPS`: Packed Single-Precision FP Square Root Reciprocal

```
RSQRTPS xmm1,xmm2/m128        ; 0F 52 /r        [KATMAI,SSE]
```

RSQRTPS computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in the source and stores the results in xmm1. The maximum error for this approximation is: |Error| <= 1.5 x 2^-12

A.5.280 RSQRTSS: Scalar Single-Precision FP Square Root Reciprocal

```
RSQRTSS xmm1,xmm2/m128          ; F3 0F 52 /r     [KATMAI,SSE]
```

RSQRTSS returns an approximation of the reciprocal of the square root of the lowest order single-precision FP value from the source, and stores it in the low doubleword of the destination register. The upper three fields of xmm1 are preserved. The maximum error for this approximation is: |Error| <= 1.5 x 2^-12

A.5.281 RSTS: Restore TSR and Descriptor

```
RSTS m80                        ; 0F 7D /0        [486,CYRIX,SMM]
```

RSTS restores Task State Register (TSR) from mem80.

A.5.282 SAHF: Store AH to Flags

```
SAHF                            ; 9E                   [8086]
```

SAHF sets the low byte of the flags word according to the contents of the AH register.

The operation of SAHF is:

```
 AH --> SF:ZF:0:AF:0:PF:1:CF
```

See also LAHF (section A.5.131).

A.5.283 SAL, SAR: Bitwise Arithmetic Shifts

```
SAL r/m8,1                      ; D0 /4                [8086]
SAL r/m8,CL                     ; D2 /4                [8086]
SAL r/m8,imm8                   ; C0 /4 ib             [186]
SAL r/m16,1                     ; o16 D1 /4            [8086]
SAL r/m16,CL                    ; o16 D3 /4            [8086]
SAL r/m16,imm8                  ; o16 C1 /4 ib         [186]
SAL r/m32,1                     ; o32 D1 /4            [386]
SAL r/m32,CL                    ; o32 D3 /4            [386]
SAL r/m32,imm8                  ; o32 C1 /4 ib         [386]

SAR r/m8,1                      ; D0 /7                [8086]
SAR r/m8,CL                     ; D2 /7                [8086]
SAR r/m8,imm8                   ; C0 /7 ib             [186]
SAR r/m16,1                     ; o16 D1 /7            [8086]
SAR r/m16,CL                    ; o16 D3 /7            [8086]
SAR r/m16,imm8                  ; o16 C1 /7 ib         [186]
SAR r/m32,1                     ; o32 D1 /7            [386]
SAR r/m32,CL                    ; o32 D3 /7            [386]
SAR r/m32,imm8                  ; o32 C1 /7 ib         [386]
```

SAL and SAR perform an arithmetic shift operation on the given source/destination (first) operand. The vacated bits are filled with zero for SAL, and with copies of the original high bit of the source operand for SAR.

SAL is a synonym for SHL (see section A.5.290). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as SHL.

The number of bits to shift by is given by the second operand. Only the bottom five bits of the shift count are considered by processors above the 8086.

You can force the longer (186 and upwards, beginning with a C1 or C0 byte) form of SAL foo,1 by using a BYTE prefix: SAL foo,BYTE 1. Similarly with SAR.

A.5.284 SALC: Set AL from Carry Flag

```
SALC                          ; D6                  [8086,UNDOC]
```

SALC is an early undocumented instruction similar in concept to SETcc (section A.5.287). Its function is to set AL to zero if the carry flag is clear, or to 0xFF if it is set.

A.5.285 SBB: Subtract with Borrow

```
SBB r/m8,reg8                 ; 18 /r                [8086]
SBB r/m16,reg16               ; o16 19 /r            [8086]
SBB r/m32,reg32               ; o32 19 /r            [386]

SBB reg8,r/m8                 ; 1A /r                [8086]
SBB reg16,r/m16               ; o16 1B /r            [8086]
SBB reg32,r/m32               ; o32 1B /r            [386]

SBB r/m8,imm8                 ; 80 /3 ib             [8086]
SBB r/m16,imm16               ; o16 81 /3 iw         [8086]
SBB r/m32,imm32               ; o32 81 /3 id         [386]

SBB r/m16,imm8                ; o16 83 /3 ib         [8086]
SBB r/m32,imm8                ; o32 83 /3 ib         [386]

SBB AL,imm8                   ; 1C ib                [8086]
SBB AX,imm16                  ; o16 1D iw            [8086]
SBB EAX,imm32                 ; o32 1D id            [386]
```

SBB performs integer subtraction: it subtracts its second operand, plus the value of the carry flag, from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent SBB instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The BYTE qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the STRICT WORD or STRICT DWORD qualifier.

To subtract one number from another without also subtracting the contents of the carry flag, use SUB (section A.5.305).

A.5.286 SCASB, SCASW, SCASD: Scan String

```
SCASB                          ; AE                     [8086]
SCASW                          ; o16 AF                 [8086]
SCASD                          ; o32 AF                 [386]
```

SCASB compares the byte in AL with the byte at [ES:DI] or [ES:EDI], and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) DI (or EDI).

The register used is DI if the address size is 16 bits, and EDI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

Segment override prefixes have no effect for this instruction: the use of ES for the load from [DI] or [EDI] cannot be overridden.

SCASW and SCASD work in the same way, but they compare a word to AX or a doubleword to EAX instead of a byte to AL, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REPE and REPNE prefixes (equivalently, REPZ and REPNZ) may be used to repeat the instruction up to CX (or ECX - again, the address size chooses which) times until the first unequal or equal element is found. To NASM, REP is an alias for REPE.

A.5.286.1 Pseudo-code examples

a16 SCASB and with Direction Flag clear (UP) is equal to

```
CMP AL, BYTE [ES:DI]
LEA DI, [DI + 1]
```

a16 REPE SCASW and with Direction Flag clear (UP) is equal to

```
JCXZ @FF
@@:
CMP AX, WORD [ES:DI]
LEA DI, [DI + 2]
a16 LOOPE @B
@@:
```

a32 SCASD with Direction Flag set (DN) is equal to

```
CMP EAX, DWORD [ES:EDI]
LEA EDI, [EDI - 4]
```

A.5.287 SETcc: Set Register from Condition

```
SETcc r/m8                     ; 0F 90+cc /2            [386]
```

SETcc sets the given 8-bit operand to zero if its condition is not satisfied, and to 1 if it is.

A.5.288 SFENCE: Store Fence

```
SFENCE                ; 0F AE /7                 [KATMAI]
```

SFENCE performs a serialising operation on all writes to memory that were issued before the

SFENCE instruction. This guarantees that all memory writes before the SFENCE instruction are visible before any writes after the SFENCE instruction.

SFENCE is ordered respective to other SFENCE instruction, MFENCE, any memory write and any other serialising instruction (such as CPUID).

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 insuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

SFENCE uses the following ModR/M encoding:

```
Mod (7:6)        = 11B
Reg/Opcode (5:3) = 111B
R/M (2:0)        = 000B
```

All other ModR/M encodings are defined to be reserved, and use of these encodings risks incompatibility with future processors.

See also LFENCE (section A.5.137) and MFENCE (section A.5.151).

A.5.289 SGDT, SIDT, SLDT: Store Descriptor Table Pointers

```
SGDT mem                      ; 0F 01 /0               [286,PRIV]
SIDT mem                      ; 0F 01 /1               [286,PRIV]
SLDT r/m16                    ; 0F 00 /0               [286,PRIV]
```

SGDT and SIDT both take a 6-byte memory area as an operand: they store the contents of the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register) as a 16-bit size limit and a 32-bit linear address into that area (limit word first, then linear address dword). The GDT and IDT instructions are the only instructions which directly use *linear* addresses, rather than segment/offset pairs.

SLDT stores the segment selector corresponding to the LDT (local descriptor table) into the given operand.

See also LGDT, LIDT and LLDT (section A.5.138).

A.5.290 SHL, SHR: Bitwise Logical Shifts

```
SHL r/m8,1                    ; D0 /4                  [8086]
SHL r/m8,CL                   ; D2 /4                  [8086]
SHL r/m8,imm8                 ; C0 /4 ib               [186]
SHL r/m16,1                   ; o16 D1 /4              [8086]
SHL r/m16,CL                  ; o16 D3 /4              [8086]
SHL r/m16,imm8                ; o16 C1 /4 ib           [186]
SHL r/m32,1                   ; o32 D1 /4              [386]
SHL r/m32,CL                  ; o32 D3 /4              [386]
SHL r/m32,imm8                ; o32 C1 /4 ib           [386]

SHR r/m8,1                    ; D0 /5                  [8086]
SHR r/m8,CL                   ; D2 /5                  [8086]
```

```
SHR r/m8,imm8                 ; C0 /5 ib             [186]
SHR r/m16,1                   ; o16 D1 /5            [8086]
SHR r/m16,CL                  ; o16 D3 /5            [8086]
SHR r/m16,imm8                ; o16 C1 /5 ib         [186]
SHR r/m32,1                   ; o32 D1 /5            [386]
SHR r/m32,CL                  ; o32 D3 /5            [386]
SHR r/m32,imm8                ; o32 C1 /5 ib         [386]
```

`SHL` and `SHR` perform a logical shift operation on the given source/destination (first) operand. The vacated bits are filled with zero.

A synonym for `SHL` is `SAL` (see section A.5.283). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as `SHL`.

The number of bits to shift by is given by the second operand. Only the bottom five bits of the shift count are considered by processors above the 8086.

You can force the longer (186 and upwards, beginning with a `C1` or `C0` byte) form of `SHL foo,1` by using a `BYTE` prefix: `SHL foo,BYTE 1`. Similarly with `SHR`.

A.5.291 `SHLD`, `SHRD`: Bitwise Double-Precision Shifts

```
SHLD r/m16,reg16,imm8         ; o16 0F A4 /r ib      [386]
SHLD r/m16,reg32,imm8         ; o32 0F A4 /r ib      [386]
SHLD r/m16,reg16,CL           ; o16 0F A5 /r         [386]
SHLD r/m16,reg32,CL           ; o32 0F A5 /r         [386]

SHRD r/m16,reg16,imm8         ; o16 0F AC /r ib      [386]
SHRD r/m32,reg32,imm8         ; o32 0F AC /r ib      [386]
SHRD r/m16,reg16,CL           ; o16 0F AD /r         [386]
SHRD r/m32,reg32,CL           ; o32 0F AD /r         [386]
```

- `SHLD` performs a double-precision left shift. It notionally places its second operand to the right of its first, then shifts the entire bit string thus generated to the left by a number of bits specified in the third operand. It then updates only the *first* operand according to the result of this. The second operand is not modified.

- `SHRD` performs the corresponding right shift: it notionally places the second operand to the *left* of the first, shifts the whole bit string right, and updates only the first operand.

For example, if `EAX` holds `0x01234567` and `EBX` holds `0x89ABCDEF`, then the instruction `SHLD EAX,EBX,4` would update `EAX` to hold `0x12345678`. Under the same conditions, `SHRD EAX,EBX,4` would update `EAX` to hold `0xF0123456`.

The number of bits to shift by is given by the third operand. Only the bottom five bits of the shift count are considered.

A.5.292 `SHUFPD`: Shuffle Packed Double-Precision FP Values

```
SHUFPD xmm1,xmm2/m128,imm8    ; 66 0F C6 /r ib  [WILLAMETTE,SSE2]
```

`SHUFPD` moves one of the packed double-precision FP values from the destination operand into the low quadword of the destination operand; the upper quadword is generated by moving one of the double-precision FP values from the source operand into the destination. The select (third) operand selects which of the values are moved to the destination 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 shuffle operand are reserved.

A.5.293 `SHUFPS`: Shuffle Packed Single-Precision FP Values

```
SHUFPS xmm1,xmm2/m128,imm8    ; 0F C6 /r ib     [KATMAI,SSE]
```

`SHUFPS` moves two of the packed single-precision FP values from the destination operand into the low quadword of the destination operand; the upper quadword is generated by moving two of the single-precision FP values from the source operand into the destination. The select (third) operand selects which of the values are moved to the destination register.

The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand 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.

A.5.294 `SMI`: System Management Interrupt

```
SMI                           ; F1                [386,UNDOC]
```

`SMI` puts some AMD processors into SMM mode. It is available on some 386 and 486 processors, and is only available when DR7 bit 12 is set, otherwise it generates an Int 1.

A.5.295 `SMINT`, `SMINTOLD`: Software SMM Entry (CYRIX)

```
SMINT                         ; 0F 38             [PENT,CYRIX]
SMINTOLD                      ; 0F 7E             [486,CYRIX]
```

`SMINT` puts the processor into SMM mode. The CPU state information is saved in the SMM memory header, and then execution begins at the SMM base address.

`SMINTOLD` is the same as `SMINT`, but was the opcode used on the 486.

This pair of opcodes is specific to the Cyrix and compatible range of processors (Cyrix, IBM, Via).

A.5.296 `SMSW`: Store Machine Status Word

```
SMSW r/m16                    ; 0F 01 /4          [286,PRIV]
```

`SMSW` stores the bottom half of the `CR0` control register (or the Machine Status Word, on 286 processors) into the destination operand. See also `LMSW` (section A.5.139).

For 32-bit code, this would store all of `CR0` in the specified register (or the bottom 16 bits if the destination is a memory location), without needing an operand size override byte.

A.5.297 `SQRTPD`: Packed Double-Precision FP Square Root

```
SQRTPD xmm1,xmm2/m128         ; 66 0F 51 /r     [WILLAMETTE,SSE2]
```

`SQRTPD` calculates the square root of the packed double-precision FP value from the source operand, and stores the double-precision results in the destination register.

A.5.298 SQRTPS: Packed Single-Precision FP Square Root

```
SQRTPS xmm1,xmm2/m128         ; 0F 51 /r        [KATMAI,SSE]
```

SQRTPS calculates the square root of the packed single-precision FP value from the source operand, and stores the single-precision results in the destination register.

A.5.299 SQRTSD: Scalar Double-Precision FP Square Root

```
SQRTSD xmm1,xmm2/m128         ; F2 0F 51 /r     [WILLAMETTE,SSE2]
```

SQRTSD calculates the square root of the low-order double-precision FP value from the source operand, and stores the double-precision result in the destination register. The high-quadword remains unchanged.

A.5.300 SQRTSS: Scalar Single-Precision FP Square Root

```
SQRTSS xmm1,xmm2/m128         ; F3 0F 51 /r     [KATMAI,SSE]
```

SQRTSS calculates the square root of the low-order single-precision FP value from the source operand, and stores the single-precision result in the destination register. The three high doublewords remain unchanged.

A.5.301 STC, STD, STI: Set Flags

```
STC                           ; F9                   [8086]
STD                           ; FD                   [8086]
STI                           ; FB                   [8086]
```

These instructions set various flags. STC sets the carry flag; STD sets the direction flag; and STI sets the interrupt flag (thus enabling interrupts).

To clear the carry, direction, or interrupt flags, use the CLC, CLD and CLI instructions (section A.5.20). To invert the carry flag, use CMC (section A.5.22).

A.5.302 STMXCSR: Store Streaming SIMD Extension Control/Status

```
STMXCSR m32                   ; 0F AE /3        [KATMAI,SSE]
```

STMXCSR stores the contents of the MXCSR control/status register to the specified memory location. MXCSR is used to enable masked/unmasked exception handling, to set rounding modes, to set flush-to-zero mode, and to view exception status flags. The reserved bits in the MXCSR register are stored as 0s.

For details of the MXCSR register, see the Intel processor docs.

See also LDMXCSR (section A.5.133).

A.5.303 STOSB, STOSW, STOSD: Store Byte to String

```
STOSB                         ; AA                   [8086]
STOSW                         ; o16 AB               [8086]
STOSD                         ; o32 AB               [386]
```

STOSB stores the byte in AL at [ES:DI] or [ES:EDI]. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) DI (or

EDI). The flags are not modified.

The register used is DI if the address size is 16 bits, and EDI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

Segment override prefixes have no effect for this instruction: the use of ES for the store to [DI] or [EDI] cannot be overridden.

STOSW and STOSD work in the same way, but they store the word in AX or the doubleword in EAX instead of the byte in AL, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REP prefix may be used to repeat the instruction CX (or ECX - again, the address size chooses which) times. The REP alias REPE, as well as the differently-encoded REPNE, are both allowed as well. They behave in the same way as REP.

A.5.303.1 Pseudo-code examples

a16 STOSB and with Direction Flag clear (UP) is equal to

```
MOV BYTE [ES:DI], AL
LEA DI, [DI + 1]
```

a16 REP STOSW and with Direction Flag clear (UP) is equal to

```
JCXZ @FF
@@:
MOV WORD [ES:DI], AX
LEA DI, [DI + 2]
a16 LOOP @B
@@:
```

a32 STOSD with Direction Flag set (DN) is equal to

```
MOV DWORD [ES:EDI], EAX
LEA EDI, [EDI - 4]
```

A.5.304 STR: Store Task Register

```
STR r/m16                     ; 0F 00 /1             [286,PRIV]
```

STR stores the segment selector corresponding to the contents of the Task Register into its operand. When the operand size is 32 bit and the destination is a register, the upper 16-bits are cleared to 0s. When the destination operand is a memory location, 16 bits are written regardless of the operand size.

A.5.305 SUB: Subtract Integers

```
SUB r/m8,reg8                 ; 28 /r                [8086]
SUB r/m16,reg16               ; o16 29 /r            [8086]
SUB r/m32,reg32               ; o32 29 /r            [386]

SUB reg8,r/m8                 ; 2A /r                [8086]
SUB reg16,r/m16               ; o16 2B /r            [8086]
SUB reg32,r/m32               ; o32 2B /r            [386]
```

```
SUB r/m8,imm8                 ; 80 /5 ib           [8086]
SUB r/m16,imm16               ; o16 81 /5 iw       [8086]
SUB r/m32,imm32               ; o32 81 /5 id       [386]

SUB r/m16,imm8                ; o16 83 /5 ib       [8086]
SUB r/m32,imm8                ; o32 83 /5 ib       [386]

SUB AL,imm8                   ; 2C ib              [8086]
SUB AX,imm16                  ; o16 2D iw          [8086]
SUB EAX,imm32                 ; o32 2D id          [386]
```

`SUB` performs integer subtraction: it subtracts its second operand from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `SBB` instruction (section A.5.285).

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The `BYTE` qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the `STRICT WORD` or `STRICT DWORD` qualifier.

The `CMP` (see section A.5.24) instruction performs the same operation as `SUB` but without writing to the destination operand. That is, it only reads the operands and writes the status flags.

A.5.306 `SUBPD`: Packed Double-Precision FP Subtract

```
SUBPD xmm1,xmm2/m128          ; 66 0F 5C /r     [WILLAMETTE,SSE2]
```

`SUBPD` subtracts the packed double-precision FP values of the source operand from those of the destination operand, and stores the result in the destination operation.

A.5.307 `SUBPS`: Packed Single-Precision FP Subtract

```
SUBPS xmm1,xmm2/m128          ; 0F 5C /r        [KATMAI,SSE]
```

`SUBPS` subtracts the packed single-precision FP values of the source operand from those of the destination operand, and stores the result in the destination operation.

A.5.308 `SUBSD`: Scalar Single-FP Subtract

```
SUBSD xmm1,xmm2/m128          ; F2 0F 5C /r     [WILLAMETTE,SSE2]
```

`SUBSD` subtracts the low-order double-precision FP value of the source operand from that of the destination operand, and stores the result in the destination operation. The high quadword is unchanged.

A.5.309 `SUBSS`: Scalar Single-FP Subtract

```
SUBSS xmm1,xmm2/m128          ; F3 0F 5C /r     [KATMAI,SSE]
```

`SUBSS` subtracts the low-order single-precision FP value of the source operand from that of the destination operand, and stores the result in the destination operation. The three high doublewords are unchanged.

A.5.310 SVDC: Save Segment Register and Descriptor

```
SVDC m80,segreg                 ; 0F 78 /r        [486,CYRIX,SMM]
```

SVDC saves a segment register (DS, ES, FS, GS, or SS) and its descriptor to mem80.

A.5.311 SVLDT: Save LDTR and Descriptor

```
SVLDT m80                       ; 0F 7A /0        [486,CYRIX,SMM]
```

SVLDT saves the Local Descriptor Table (LDTR) to mem80.

A.5.312 SVTS: Save TSR and Descriptor

```
SVTS m80                        ; 0F 7C /0        [486,CYRIX,SMM]
```

SVTS saves the Task State Register (TSR) to mem80.

A.5.313 SYSCALL: Call Operating System

```
SYSCALL                         ; 0F 05                [P6,AMD]
```

SYSCALL provides a fast method of transferring control to a fixed entry point in an operating system.

- The EIP register is copied into the ECX register.
- Bits [31-0] of the 64-bit SYSCALL/SYSRET Target Address Register (STAR) are copied into the EIP register.
- Bits [47-32] of the STAR register specify the selector that is copied into the CS register.
- Bits [47-32]+1000b of the STAR register specify the selector that is copied into the SS register.

The CS and SS registers should not be modified by the operating system between the execution of the SYSCALL instruction and its corresponding SYSRET instruction.

For more information, see the SYSCALL and SYSRET Instruction Specification (AMD document number 21086.pdf).

A.5.314 SYSENTER: Fast System Call

```
SYSENTER                        ; 0F 34                [P6]
```

SYSENTER executes a fast call to a level 0 system procedure or routine. Before using this instruction, various MSRs need to be set up:

- SYSENTER_CS_MSR contains the 32-bit segment selector for the privilege level 0 code segment. (This value is also used to compute the segment selector of the privilege level 0 stack segment.)
- SYSENTER_EIP_MSR contains the 32-bit offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
- SYSENTER_ESP_MSR contains the 32-bit stack pointer for the privilege level 0 stack.

SYSENTER performs the following sequence of operations:

- Loads the segment selector from the SYSENTER_CS_MSR into the CS register.
- Loads the instruction pointer from the SYSENTER_EIP_MSR into the EIP register.
- Adds 8 to the value in SYSENTER_CS_MSR and loads it into the SS register.
- Loads the stack pointer from the SYSENTER_ESP_MSR into the ESP register.
- Switches to privilege level 0.
- Clears the VM flag in the EFLAGS register, if the flag is set.
- Begins executing the selected system procedure.

In particular, note that this instruction des not save the values of CS or (E)IP. If you need to return to the calling code, you need to write your code to cater for this.

For more information, see the Intel Architecture Software Developer's Manual, Volume 2.

A.5.315 SYSEXIT: Fast Return From System Call

```
SYSEXIT                       ; 0F 35                [P6,PRIV]
```

SYSEXIT executes a fast return to privilege level 3 user code. This instruction is a companion instruction to the SYSENTER instruction, and can only be executed by privilege level 0 code. Various registers need to be set up before calling this instruction:

- SYSENTER_CS_MSR contains the 32-bit segment selector for the privilege level 0 code segment in which the processor is currently executing. (This value is used to compute the segment selectors for the privilege level 3 code and stack segments.)
- EDX contains the 32-bit offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
- ECX contains the 32-bit stack pointer for the privilege level 3 stack.

SYSEXIT performs the following sequence of operations:

- Adds 16 to the value in SYSENTER_CS_MSR and loads the sum into the CS selector register.
- Loads the instruction pointer from the EDX register into the EIP register.
- Adds 24 to the value in SYSENTER_CS_MSR and loads the sum into the SS selector register.
- Loads the stack pointer from the ECX register into the ESP register.
- Switches to privilege level 3.
- Begins executing the user code at the EIP address.

For more information on the use of the SYSENTER and SYSEXIT instructions, see the Intel Architecture Software Developer's Manual, Volume 2.

A.5.316 SYSRET: Return From Operating System

```
SYSRET                        ; 0F 07                [P6,AMD,PRIV]
```

SYSRET is the return instruction used in conjunction with the SYSCALL instruction to provide fast entry/exit to an operating system.

- The ECX register, which points to the next sequential instruction after the corresponding SYSCALL instruction, is copied into the EIP register.
- Bits [63-48] of the STAR register specify the selector that is copied into the CS register.
- Bits [63-48]+1000b of the STAR register specify the selector that is copied into the SS register.
- Bits [1-0] of the SS register are set to 11b (RPL of 3) regardless of the value of bits [49-48] of the STAR register.

The CS and SS registers should not be modified by the operating system between the execution of the SYSCALL instruction and its corresponding SYSRET instruction.

For more information, see the SYSCALL and SYSRET Instruction Specification (AMD document number 21086.pdf).

A.5.317 TEST: Test Bits (notional bitwise AND)

```
TEST r/m8,reg8                ; 84 /r                [8086]
TEST r/m16,reg16              ; o16 85 /r            [8086]
TEST r/m32,reg32              ; o32 85 /r            [386]

TEST r/m8,imm8                ; F6 /0 ib             [8086]
TEST r/m16,imm16              ; o16 F7 /0 iw         [8086]
TEST r/m32,imm32              ; o32 F7 /0 id         [386]

TEST AL,imm8                  ; A8 ib                [8086]
TEST AX,imm16                 ; o16 A9 iw            [8086]
TEST EAX,imm32                ; o32 A9 id            [386]
```

TEST performs a 'mental' bitwise AND of its two operands, and affects the flags as if the operation had taken place, but does not store the result of the operation anywhere. (For bitwise AND that does store the result, see section A.5.8.)

The Carry Flag is cleared by TEST. The Zero Flag is set according to whether the result is zero.

The order of operands to TEST does not matter, except that an immediate operand must always be used as the second operand (source operand). So, when a memory operand and a register operand are used together, the memory operand may be either the first or the second operand in the source text without a discernible difference. Two-register instructions may choose either of the two registers for the ModR/M operand and the other for the /r operand. This choice may result in differing disassembly.

A.5.318 `UCOMISD`: Unordered Scalar Double-Precision FP compare and set EFLAGS

```
UCOMISD xmm1,xmm2/m128          ; 66 0F 2E /r     [WILLAMETTE,SSE2]
```

`UCOMISD` compares the low-order double-precision FP numbers in the two operands, and sets the `ZF`, `PF` and `CF` bits in the `EFLAGS` register. In addition, the `OF`, `SF` and `AF` bits in the `EFLAGS` register are zeroed out. The unordered predicate (`ZF`, `PF` and `CF` all set) is returned if either source operand is a `NaN` (`qNaN` or `sNaN`).

A.5.319 `UCOMISS`: Unordered Scalar Single-Precision FP compare and set EFLAGS

```
UCOMISS xmm1,xmm2/m128          ; 0F 2E /r        [KATMAI,SSE]
```

`UCOMISS` compares the low-order single-precision FP numbers in the two operands, and sets the `ZF`, `PF` and `CF` bits in the `EFLAGS` register. In addition, the `OF`, `SF` and `AF` bits in the `EFLAGS` register are zeroed out. The unordered predicate (`ZF`, `PF` and `CF` all set) is returned if either source operand is a `NaN` (`qNaN` or `sNaN`).

A.5.320 `UD0`, `UD1`, `UD2`: Undefined Instruction

```
UD0                             ; 0F FF                [186,UNDOC]
UD1                             ; 0F B9                [186,UNDOC]
UD2                             ; 0F 0B                [186]
```

`UDx` can be used to generate an invalid opcode exception, for testing purposes.

`UD0` is specifically documented by AMD as being reserved for this purpose.

`UD1` is documented by Intel as being available for this purpose.

`UD2` is specifically documented by Intel as being reserved for this purpose. Intel document this as the preferred method of generating an invalid opcode exception.

All these opcodes can be used to generate invalid opcode exceptions on all currently available processors.

A.5.321 `UMOV`: User Move Data

```
UMOV r/m8,reg8                  ; 0F 10 /r             [386,UNDOC]
UMOV r/m16,reg16                ; o16 0F 11 /r         [386,UNDOC]
UMOV r/m32,reg32                ; o32 0F 11 /r         [386,UNDOC]

UMOV reg8,r/m8                  ; 0F 12 /r             [386,UNDOC]
UMOV reg16,r/m16                ; o16 0F 13 /r         [386,UNDOC]
UMOV reg32,r/m32                ; o32 0F 13 /r         [386,UNDOC]
```

This undocumented instruction is used by in-circuit emulators to access user memory (as opposed to host memory). It is used just like an ordinary memory/register or register/register `MOV` instruction, but accesses user space.

This instruction is only available on some AMD and IBM 386 and 486 processors.

A.5.322 `UNPCKHPD`: Unpack and Interleave High Packed Double-Precision FP Values

```
UNPCKHPD xmm1,xmm2/m128       ; 66 0F 15 /r     [WILLAMETTE,SSE2]
```

`UNPCKHPD` performs an interleaved unpack of the high-order data elements of the source and destination operands, saving the result in `xmm1`. It ignores the lower half of the sources.

The operation of this instruction is:

```
dst[63-0]   := dst[127-64];
dst[127-64] := src[127-64].
```

A.5.323 `UNPCKHPS`: Unpack and Interleave High Packed Single-Precision FP Values

```
UNPCKHPS xmm1,xmm2/m128       ; 0F 15 /r        [KATMAI,SSE]
```

`UNPCKHPS` performs an interleaved unpack of the high-order data elements of the source and destination operands, saving the result in `xmm1`. It ignores the lower half of the sources.

The operation of this instruction is:

```
dst[31-0]   := dst[95-64];
dst[63-32]  := src[95-64];
dst[95-64]  := dst[127-96];
dst[127-96] := src[127-96].
```

A.5.324 `UNPCKLPD`: Unpack and Interleave Low Packed Double-Precision FP Data

```
UNPCKLPD xmm1,xmm2/m128       ; 66 0F 14 /r     [WILLAMETTE,SSE2]
```

`UNPCKLPD` performs an interleaved unpack of the low-order data elements of the source and destination operands, saving the result in `xmm1`. It ignores the lower half of the sources.

The operation of this instruction is:

```
dst[63-0]   := dst[63-0];
dst[127-64] := src[63-0].
```

A.5.325 `UNPCKLPS`: Unpack and Interleave Low Packed Single-Precision FP Data

```
UNPCKLPS xmm1,xmm2/m128       ; 0F 14 /r        [KATMAI,SSE]
```

`UNPCKLPS` performs an interleaved unpack of the low-order data elements of the source and destination operands, saving the result in `xmm1`. It ignores the lower half of the sources.

The operation of this instruction is:

```
dst[31-0]   := dst[31-0];
dst[63-32]  := src[31-0];
dst[95-64]  := dst[63-32];
dst[127-96] := src[63-32].
```

A.5.326 `VERR`, `VERW`: Verify Segment Readability/Writability

```
VERR r/m16                    ; 0F 00 /4             [286,PRIV]

VERW r/m16                    ; 0F 00 /5             [286,PRIV]
```

- `VERR` sets the zero flag if the segment specified by the selector in its operand can be read from at the current privilege level. Otherwise it is cleared.
- `VERW` sets the zero flag if the segment can be written.

A.5.327 `WAIT`: Wait for Floating-Point Processor

```
WAIT                          ; 9B                   [8086]
FWAIT                         ; 9B                   [8086]
```

`WAIT`, on 8086 systems with a separate 8087 FPU, waits for the FPU to have finished any operation it is engaged in before continuing main processor operations, so that (for example) an FPU store to main memory can be guaranteed to have completed before the CPU tries to read the result back out.

On higher processors, `WAIT` is unnecessary for this purpose, and it has the alternative purpose of ensuring that any pending unmasked FPU exceptions have happened before execution continues.

A.5.328 `WBINVD`: Write Back and Invalidate Cache

```
WBINVD                        ; 0F 09                [486]
```

`WBINVD` invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It writes the contents of the caches back to memory first, so no data is lost. To flush the caches quickly without bothering to write the data back first, use `INVD` (section A.5.125).

A.5.329 `WRMSR`: Write Model-Specific Registers

```
WRMSR                         ; 0F 30                [PENT]
```

`WRMSR` writes the value in `EDX:EAX` to the processor Model-Specific Register (MSR) whose index is stored in `ECX`. See also `RDMSR` (section A.5.270).

A.5.330 `WRSHR`: Write SMM Header Pointer Register

```
WRSHR r/m32                   ; 0F 37 /0        [386,CYRIX,SMM]
```

`WRSHR` loads the contents of either a 32-bit memory location or a 32-bit register into the SMM header pointer register.

See also `RDSHR` (section A.5.272).

A.5.331 `XADD`: Exchange and Add

```
XADD r/m8,reg8                ; 0F C0 /r             [486]
XADD r/m16,reg16              ; o16 0F C1 /r         [486]
XADD r/m32,reg32              ; o32 0F C1 /r         [486]
```

`XADD` exchanges the values in its two operands, and then adds them together and writes the result into the destination (first) operand. This instruction can be used with a LOCK prefix for multi-processor synchronisation purposes.

A.5.332 XBTS: Extract Bit String

```
XBTS reg16,r/m16                ; o16 0F A6 /r          [386,UNDOC]
XBTS reg32,r/m32                ; o32 0F A6 /r          [386,UNDOC]
```

The implied operation of this instruction is:

```
XBTS r/m16,reg16,AX,CL
XBTS r/m32,reg32,EAX,CL
```

Writes a bit string from the source operand to the destination. CL indicates the number of bits to be copied, and (E)AX indicates the low order bit offset in the source. The bits are written to the low order bits of the destination register. For example, if CL is set to 4 and AX (for 16-bit code) is set to 5, bits 5-8 of src will be copied to bits 0-3 of dst. This instruction is very poorly documented, and I have been unable to find any official source of documentation on it.

XBTS is supported only on the early Intel 386s, and conflicts with the opcodes for CMPXCHG486 (on early Intel 486s). NASM supports it only for completeness. Its counterpart is IBTS (see section A.5.116).

A.5.333 XCHG: Exchange

```
XCHG reg8,r/m8                  ; 86 /r                 [8086]
XCHG reg16,r/m16                ; o16 87 /r             [8086]
XCHG reg32,r/m32                ; o32 87 /r             [386]

XCHG r/m8,reg8                  ; 86 /r                 [8086]
XCHG r/m16,reg16                ; o16 87 /r             [8086]
XCHG r/m32,reg32                ; o32 87 /r             [386]

XCHG AX,reg16                   ; o16 90+r              [8086]
XCHG EAX,reg32                  ; o32 90+r              [386]
XCHG reg16,AX                   ; o16 90+r              [8086]
XCHG reg32,EAX                  ; o32 90+r              [386]
```

XCHG exchanges the values in its two operands. It can be used with a LOCK prefix for purposes of multi-processor synchronisation.

XCHG AX,AX or XCHG EAX,EAX (depending on the BITS setting) generates the opcode 90h, and so is a synonym for NOP (section A.5.190).

The XCHG encodings as listed are actually all listed twice. That is because the order of operands does not matter, so a memory operand or the AX or EAX operand may be either the first or the second operand in the source text without a discernible difference. Two-register instructions without an AX or EAX operand may choose either of the two registers for the ModR/M operand and the other for the /r operand. This choice may result in differing disassembly.

A.5.334 XLATB: Translate Byte in Lookup Table

```
XLAT                            ; D7                    [8086]
XLATB                           ; D7                    [8086]
```

XLATB adds the value in AL, treated as an unsigned byte, to BX or EBX, and loads the byte from the resulting address (in the segment specified by DS) back into AL.

The base register used is BX if the address size is 16 bits, and EBX if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

On 386 or higher level machines, the segment register used to load from [BX+AL] or [EBX+AL] can be overridden by using a segment register name as a prefix (for example, ES XLATB). It is reported that a segment override may be ignored by CPUs of a lower level than a 386.

A.5.335 XOR: Bitwise Exclusive OR

```
XOR r/m8,reg8                 ; 30 /r                [8086]
XOR r/m16,reg16               ; o16 31 /r            [8086]
XOR r/m32,reg32               ; o32 31 /r            [386]

XOR reg8,r/m8                 ; 32 /r                [8086]
XOR reg16,r/m16               ; o16 33 /r            [8086]
XOR reg32,r/m32               ; o32 33 /r            [386]

XOR r/m8,imm8                 ; 80 /6 ib             [8086]
XOR r/m16,imm16               ; o16 81 /6 iw         [8086]
XOR r/m32,imm32               ; o32 81 /6 id         [386]

XOR r/m16,imm8                ; o16 83 /6 ib         [8086]
XOR r/m32,imm8                ; o32 83 /6 ib         [386]

XOR AL,imm8                   ; 34 ib                [8086]
XOR AX,imm16                  ; o16 35 iw            [8086]
XOR EAX,imm32                 ; o32 35 id            [386]
```

XOR performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. The BYTE qualifier can be used to force NASM to generate this form of the instruction. Recent versions of NASM automatically optimise to this form if the immediate operand's value is known during the assembling of that instruction, and fits in the range of a signed byte. The longer variant can then still be forced using the STRICT WORD or STRICT DWORD qualifier.

The Carry Flag is cleared by XOR. The Zero Flag is set according to whether the result is zero.

The MMX instruction PXOR (see section A.5.266) performs the same operation on the 64-bit MMX registers.

A.5.336 XORPD: Bitwise Logical XOR of Double-Precision FP Values

```
XORPD xmm1,xmm2/m128          ; 66 0F 57 /r     [WILLAMETTE,SSE2]
```

XORPD returns a bit-wise logical XOR between the source and destination operands, storing the result in the destination operand.

A.5.337 `XORPS`: Bitwise Logical XOR of Single-Precision FP Values

```
XORPS xmm1,xmm2/m128          ; 0F 57 /r        [KATMAI,SSE]
```

`XORPS` returns a bit-wise logical XOR between the source and destination operands, storing the result in the destination operand.

Source Control Revision ID

hg 10e2ef760411, from commit on at 2025-08-28 13:43:46 +0200

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/insref/rev/10e2ef760411

Index