Appendix A: Intel 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 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://www.intel.com/`.

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, and `reg32` a 32-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. In addition, some registers (such as `AL`, `DX` or `ECX`) 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 needs an explicit specifier: for example, `ADD ESP,16` could be interpreted as either `ADD r/m32,imm32` or `ADD r/m32,imm8`. NASM chooses the former by default, and so you must specify `ADD ESP,BYTE 16` for the latter.

- 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` and `memoffs32`.

- Register or memory choices: many instructions can accept either a register *or* a memory reference as an operand. `r/m8` is a shorthand for `reg8/mem8`; similarly `r/m16` and `r/m32`. `r/m64` is MMX-related, and is a shorthand for `mmxreg/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 `40+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 `40+cc`, when the instruction contains the `NE` condition, generates the hex byte `45`. Condition codes and their numeric representations are given in section A.2.2.
- 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.2.3.
- 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.2.3; 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.

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.

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.
- 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.
- 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.

- 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.
- Control registers: CR0 is 0, CR2 is 2, CR3 is 3, and CR4 is 4.
- 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.

A.2.3 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.

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.

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.

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).
- 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 4, 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).

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.
- 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).
- `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.
- `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.

A.4 `AAA`, `AAS`, `AAM`, `AAD`: ASCII Adjustments

```
AAA                           ; 37                    [8086]

AAS                           ; 3F                    [8086]

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

AAM                           ; D4 0A                 [8086]
AAM imm                       ; 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.23.

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

`AAM` 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`.

`AAD` 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.

A.5 `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 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. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

To add two numbers without also adding the contents of the carry flag, use `ADD` (section A.6).

A.6 `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 /0 ib               [8086]
ADD r/m16,imm16               ; o16 81 /0 iw           [8086]
ADD r/m32,imm32               ; o32 81 /0 id           [386]

ADD r/m16,imm8                ; o16 83 /0 ib           [8086]
ADD r/m32,imm8                ; o32 83 /0 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 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 (section A.5).

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. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

A.7 `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.

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. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction PAND (see section A.116) performs the same operation on the 64-bit MMX registers.

A.8 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.9 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.10 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 a set bit in its source (second) operand, starting from the bottom, 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.

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).

A.11 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.

A.12 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,imm                 ; o16 0F BA /5 ib       [386]
BTS r/m32,imm                 ; 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 bit offset should be no greater than the size of the operand.

A.13 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.

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`.

The CALL r/m forms given above are near calls; NASM will accept the NEAR keyword (e.g. CALL NEAR [address]), even though it is not strictly necessary.

A.14 CBW, CWD, CDQ, CWDE: Sign Extensions

```
CBW                             ; o16 98                [8086]
CWD                             ; o16 99                [8086]
CDQ                             ; o32 99                [386]
CWDE                            ; o32 98                [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. CWD extends AX into DX:AX by repeating the top bit of AX throughout DX. CWDE extends AX into EAX, and CDQ extends EAX into EDX:EAX.

A.15 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.156). To invert the carry flag, use CMC (section A.16).

A.16 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.17 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 above, they may not be supported by all Pentium Pro processors; the CPUID instruction (section A.22) will return a bit which indicates whether conditional moves are supported.

A.18 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 /0 ib             [8086]
CMP r/m16,imm16               ; o16 81 /0 iw         [8086]
CMP r/m32,imm32               ; o32 81 /0 id         [386]

CMP r/m16,imm8                ; o16 83 /0 ib         [8086]
CMP r/m32,imm8                ; o32 83 /0 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.

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. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

A.19 `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 byte is found.

A.20 `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 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 leaves the destination alone.

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.21 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 leaves the memory area untouched.

A.22 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 serialise 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, described in section 3.4.2), 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.21) is supported, bit 15 is set if the conditional move instructions (section A.17 and section A.34) 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 on Intel's web site.

A.23 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.4.

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.24 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.159). See also INC (section A.79).

A.25 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.76.

A.26 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.27 ENTER: Create Stack Frame

```
ENTER imm,imm                ; 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
          SUB ESP,operand1   ; or SUB SP,operand1 in 16 bits
```

This creates a stack frame with the procedure parameters accessible upwards from EBP, and local variables accessible downwards from EBP.

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.

Stack frames created by ENTER can be destroyed by the LEAVE instruction: see section A.94.

A.28 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 to +1.

A.29 FABS: Floating-Point Absolute Value

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

FABS computes the absolute value of ST0, storing the result back in ST0.

A.30 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.

A.31 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.32 FCHS: Floating-Point Change Sign

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

FCHS negates the number in ST0: negative numbers become positive, and vice versa.

A.33 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.34 `FCMOVcc`: Floating-Point Conditional Move

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

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

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

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

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

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

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

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

The `FCMOV` 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.22) will return a bit which indicates whether conditional moves are supported.

A.35 `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.69) 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.36 FCOS: Cosine

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

FCOS computes the cosine of ST0 (in radians), and stores the result in ST0. See also FSINCOS (section A.61).

A.37 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.46).

A.38 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.39 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 /0                  [8086,FPU]
FDIVR mem64                   ; DC /0                  [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.

A.40 FFREE: Flag Floating-Point Register as Unused

```
FFREE fpureg                  ; DD C0+r                [8086,FPU]
```

FFREE marks the given register as being empty.

A.41 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.42 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.43 FIDIV, FIDIVR: Floating-Point/Integer Division

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

FIDIVR mem16                  ; DE /0                  [8086,FPU]
FIDIVR mem32                  ; DA /0                  [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.44 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 /0                [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.45 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.46 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.37).

A.47 FINIT, FNINIT: Initialise Floating-Point Unit

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

FINIT initialises the FPU to its default state. It flags all registers as empty, though it does not actually change their values. FNINIT does the same, without first waiting for pending exceptions to clear.

A.48 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.49 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.50 `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. `FLD1` pushes the value 1; `FLDL2E` pushes the base-2 logarithm of e; `FLDL2T` pushes the base-2 log of 10; `FLDLG2` pushes the base-10 log of 2; `FLDLN2` pushes the base-e log of 2; `FLDPI` pushes pi; and `FLDZ` pushes zero.

A.51 `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.64).

A.52 `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.65).

A.53 `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.54 `FNOP`: Floating-Point No Operation

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

`FNOP` does nothing.

A.55 `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.

A.56 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.57 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.58 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.65) 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.59 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.60 FSETPM: Set Protected Mode

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

This instruction initalises 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.61 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.36) in succession.

A.62 FSQRT: Floating-Point Square Root

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

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

A.63 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 /0                  [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.64 FSTCW: Store Floating-Point Control Word

```
FSTCW mem16                   ; 9B D9 /0               [8086,FPU]
FNSTCW mem16                  ; D9 /0                  [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.51).

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

A.65 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.52).

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

A.66 FSTSW: Store Floating-Point Status Word

```
FSTSW mem16                   ; 9B DD /0               [8086,FPU]
FSTSW AX                      ; 9B DF E0               [286,FPU]

FNSTSW mem16                  ; DD /0                  [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.67 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.68 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.69 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.35) 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.70 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: 000 (respectively) for an unsupported format, 001 for a NaN, 010 for a normal finite number, 011 for an infinity, 100 for a zero, 101 for an empty register, and 110 for a denormal. It also sets the C1 flag to the sign of the number.

A.71 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.72 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.73 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.74 HLT: Halt Processor

```
HLT                            ; F4                     [8086]
```

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

A.75 IBTS: Insert Bit String

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

No clear documentation seems to be available for this instruction: the best I've been able to find reads 'Takes a string of bits from the second operand and puts them in the first operand'. It is present only in early 386 processors, and conflicts with the opcodes for `CMPXCHG486`. NASM supports it only for completeness. Its counterpart is `XBTS` (see section A.167).

A.76 `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.25.

A.77 `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           [286]
IMUL reg16,imm16              ; o16 69 /r iw           [286]
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           [286]
IMUL reg16,r/m16,imm16        ; o16 69 /r iw           [286]
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 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. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

Unsigned integer multiplication is performed by the MUL instruction: see section A.107.

A.78 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.111).

A.79 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.6). See also DEC (section A.24).

A.80 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 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.

See also OUTSB, OUTSW and OUTSD (section A.112).

A.81 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.82) instead.

A.82 INT3, INT1, ICEBP, INT01: Breakpoints

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

INT3                          ; CC                     [8086]
```

INT1 and INT3 are short one-byte forms of the instructions INT 1 and INT 3 (see section A.81). 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 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.83 INTO: Interrupt if Overflow

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

INTO performs an INT 4 software interrupt (see section A.81) if and only if the overflow flag is set.

A.84 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.164).

A.85 INVLPG: Invalidate TLB Entry

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

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

A.86 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.87 JCXZ, JECXZ: Jump if CX/ECX Zero

```
JCXZ imm                      ; o16 E3 rb            [8086]
JECXZ imm                     ; o32 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.88 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 mem                   ; 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. NASM does not choose when to generate JMP SHORT for you: you must explicitly code SHORT every time you want 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.

The CALL r/m forms given above are near calls; NASM will accept the NEAR keyword (e.g. CALL NEAR [address]), even though it is not strictly necessary.

A.89 Jcc: Conditional Branch

```
Jcc imm                       ; 70+cc rb             [8086]
Jcc NEAR imm                  ; 0F 80+cc rw/rd       [386]
```

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 NEAR form is a 386 extension to the instruction set, and can span the full size of a segment. NASM will not override your choice of jump instruction: if you want Jcc NEAR, you have to use the NEAR keyword.

The SHORT keyword is allowed on the first form of the instruction, for clarity, but is not necessary.

A.90 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. See also SAHF (section A.145).

A.91 LAR: Load Access Rights

```
LAR reg16,r/m16                ; o16 0F 02 /r           [286,PRIV]
LAR reg32,r/m32                ; o32 0F 02 /r           [286,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.92 LDS, LES, LFS, LGS, LSS: Load Far Pointer

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

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

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.93 LEA: Load Effective Address

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

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.

A.94 LEAVE: Destroy Stack Frame

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

LEAVE destroys a stack frame of the form created by the ENTER instruction (see section A.27). 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.95 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 32-bit linear address and a 16-bit size limit from that area (in the opposite order) into the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register). These 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.151).

A.96 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.155).

A.97 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.98 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 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.

A.99 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.100 `LSL`: Load Segment Limit

```
LSL reg16,r/m16               ; o16 0F 03 /r           [286,PRIV]
LSL reg32,r/m32               ; o32 0F 03 /r           [286,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.101 `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.102 `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.

CR4 is only a supported register on the Pentium and above.

A.103 MOVD: Move Doubleword to/from MMX Register

```
MOVD mmxreg,r/m32             ; 0F 6E /r                [PENT,MMX]
MOVD r/m32,mmxreg             ; 0F 7E /r                [PENT,MMX]
```

MOVD copies 32 bits from its source (second) operand into its destination (first) operand. When the destination is a 64-bit MMX register, the top 32 bits are set to zero.

A.104 MOVQ: Move Quadword to/from MMX Register

```
MOVQ mmxreg,r/m64             ; 0F 6F /r                [PENT,MMX]
MOVQ r/m64,mmxreg             ; 0F 7F /r                [PENT,MMX]
```

MOVQ copies 64 bits from its source (second) operand into its destination (first) operand.

A.105 MOVSB, MOVSW, MOVSD: Move String

```
MOVSB                         ; A4                      [8086]
MOVSW                         ; o16 A5                  [8086]
MOVSD                         ; o32 A5                  [386]
```

MOVSB copies the byte at [ES:DI] or [ES:EDI] to [DS:SI] or [DS:ESI]. 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 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.

A.106 `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.107 `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.77.

A.108 `NEG`, `NOT`: Two’s and One’s 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 one’s complement (inverts all the bits).

A.109 `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.168).

A.110 `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. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `POR` (see section A.129) performs the same operation on the 64-bit MMX registers.

A.111 `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]
```

`IN` 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.78).

A.112 `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 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.

A.113 PACKSSDW, PACKSSWB, PACKUSWB: Pack Data

```
PACKSSDW mmxreg,r/m64         ; 0F 6B /r               [PENT,MMX]
PACKSSWB mmxreg,r/m64         ; 0F 63 /r               [PENT,MMX]
PACKUSWB mmxreg,r/m64         ; 0F 67 /r               [PENT,MMX]
```

All these instructions start by forming a notional 128-bit word by placing the source (second) operand on the left of the destination (first) operand. `PACKSSDW` then splits this 128-bit word into four doublewords, converts each to a word, and loads them side by side into the destination register; `PACKSSWB` and `PACKUSWB` both split the 128-bit word into eight words, converts each to a byte, and loads *those* side by side into the destination register.

`PACKSSDW` and `PACKSSWB` perform signed saturation when reducing the length of numbers: if the number is too large to fit into the reduced space, they replace it by the largest signed number (`7FFFh` or `7Fh`) that *will* fit, and if it is too small then they replace it by the smallest signed number (`8000h` or `80h`) that will fit. `PACKUSWB` performs unsigned saturation: it treats its input as unsigned, and replaces it by the largest unsigned number that will fit.

A.114 PADDxx: MMX Packed Addition

```
PADDB mmxreg,r/m64            ; 0F FC /r               [PENT,MMX]
PADDW mmxreg,r/m64            ; 0F FD /r               [PENT,MMX]
PADDD mmxreg,r/m64            ; 0F FE /r               [PENT,MMX]

PADDSB mmxreg,r/m64           ; 0F EC /r               [PENT,MMX]
PADDSW mmxreg,r/m64           ; 0F ED /r               [PENT,MMX]

PADDUSB mmxreg,r/m64          ; 0F DC /r               [PENT,MMX]
PADDUSW mmxreg,r/m64          ; 0F DD /r               [PENT,MMX]
```

`PADDxx` all perform packed addition between their two 64-bit operands, storing the result in the destination (first) operand. The `PADDxB` forms treat the 64-bit operands as vectors of eight bytes, and add each byte individually; `PADDxW` treat the operands as vectors of four words; and `PADDD` treats its operands as vectors of two doublewords.

`PADDSB` and `PADDSW` perform signed saturation on the sum of each pair of bytes or words: if the result of an addition is too large or too small to fit into a signed byte or word result, it is clipped (saturated) to the largest or smallest value which *will* fit. `PADDUSB` and `PADDUSW` similarly perform unsigned saturation, clipping to `0FFh` or `0FFFFh` if the result is larger than that.

A.115 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 not placed in the register specified by the first operand, but instead in the register whose number differs from the first operand only in the last bit. So `PADDSIW MM0,MM2` would put the result in `MM1`, but `PADDSIW MM1,MM2` would put the result in `MM0`.

A.116 PAND, PANDN: MMX Bitwise AND and AND-NOT

```
PAND mmxreg,r/m64             ; 0F DB /r               [PENT,MMX]
PANDN mmxreg,r/m64            ; 0F DF /r               [PENT,MMX]
```

`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 one's complement operation on the destination (first) operand first.

A.117 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.

A.118 PCMPxx: MMX Packed Comparison

```
PCMPEQB mmxreg,r/m64          ; 0F 74 /r              [PENT,MMX]
PCMPEQW mmxreg,r/m64          ; 0F 75 /r              [PENT,MMX]
PCMPEQD mmxreg,r/m64          ; 0F 76 /r              [PENT,MMX]

PCMPGTB mmxreg,r/m64          ; 0F 64 /r              [PENT,MMX]
PCMPGTW mmxreg,r/m64          ; 0F 65 /r              [PENT,MMX]
PCMPGTD mmxreg,r/m64          ; 0F 66 /r              [PENT,MMX]
```

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 eight bytes, PCMPxxW treats them as vectors of four words, and PCMPxxD as two 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.119 PDISTIB: MMX Packed Distance and Accumulate with Implied Register

```
PDISTIB mmxreg,mem64          ; 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.

The implied output register is found in the same way as PADDSIW (section A.115).

Note that PDISTIB cannot take a register as its second source operand.

A.120 PMACHRIW: MMX Packed Multiply and Accumulate with Rounding

```
PMACHRIW mmxreg,mem64         ; 0F 5E /r              [CYRIX,MMX]
```

PMACHRIW acts almost identically to PMULHRIW (section A.123), but instead of *storing* its result in the implied destination register, it *adds* its result, as four packed words, to the implied destination register. No saturation is done: the addition can wrap around.

Note that PMACHRIW cannot take a register as its second source operand.

A.121 PMADDWD: MMX Packed Multiply and Add

```
PMADDWD mmxreg,r/m64          ; 0F F5 /r              [PENT,MMX]
```

PMADDWD treats its two inputs as vectors of four signed words. It multiplies corresponding elements of the two operands, giving four signed doubleword results. The top two of these are added and placed in the top 32 bits of the destination (first) operand; the bottom two are added and placed in the bottom 32 bits.

A.122 PMAGW: MMX Packed Magnitude

```
PMAGW mmxreg,r/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.123 PMULHRW, PMULHRIW: MMX Packed Multiply High with Rounding

```
PMULHRW mmxreg,r/m64          ; 0F 59 /r             [CYRIX,MMX]
PMULHRIW mmxreg,r/m64         ; 0F 5D /r             [CYRIX,MMX]
```

These instructions, specific to the Cyrix MMX extensions, treat their operands as vectors of four signed words. Words in corresponding positions are multiplied, to give a 32-bit value in which bits 30 and 31 are guaranteed equal. Bits 30 to 15 of this value (bit mask 0x7FFF8000) are taken and stored in the corresponding position of the destination operand, after first rounding the low bit (equivalent to adding 0x4000 before extracting bits 30 to 15).

For PMULHRW, the destination operand is the first operand; for PMULHRIW the destination operand is implied by the first operand in the manner of PADDSIW (section A.115).

A.124 PMULHW, PMULLW: MMX Packed Multiply

```
PMULHW mmxreg,r/m64           ; 0F E5 /r             [PENT,MMX]
PMULLW mmxreg,r/m64           ; 0F D5 /r             [PENT,MMX]
```

PMULxW treats its two inputs as vectors of four signed words. It multiplies corresponding elements of the two operands, giving four signed 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.125 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.115).

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, and 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.126 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.

A.127 `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.135), 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.128 `POPFx`: Pop Flags Register

```
POPF                          ; 9D                    [186]
POPFW                         ; o16 9D                [186]
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.136).

A.129 `POR`: MMX Bitwise OR

```
POR mmxreg,r/m64              ; 0F EB /r              [PENT,MMX]
```

`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.130 `PSLLx`, `PSRLx`, `PSRAx`: MMX Bit Shifts

```
PSLLW mmxreg,r/m64            ; 0F F1 /r             [PENT,MMX]
PSLLW mmxreg,imm8             ; 0F 71 /6 ib          [PENT,MMX]

PSLLD mmxreg,r/m64            ; 0F F2 /r             [PENT,MMX]
PSLLD mmxreg,imm8             ; 0F 72 /6 ib          [PENT,MMX]

PSLLQ mmxreg,r/m64            ; 0F F3 /r             [PENT,MMX]
PSLLQ mmxreg,imm8             ; 0F 73 /6 ib          [PENT,MMX]

PSRAW mmxreg,r/m64            ; 0F E1 /r             [PENT,MMX]
PSRAW mmxreg,imm8             ; 0F 71 /4 ib          [PENT,MMX]

PSRAD mmxreg,r/m64            ; 0F E2 /r             [PENT,MMX]
PSRAD mmxreg,imm8             ; 0F 72 /4 ib          [PENT,MMX]

PSRLW mmxreg,r/m64            ; 0F D1 /r             [PENT,MMX]
PSRLW mmxreg,imm8             ; 0F 71 /2 ib          [PENT,MMX]

PSRLD mmxreg,r/m64            ; 0F D2 /r             [PENT,MMX]
PSRLD mmxreg,imm8             ; 0F 72 /2 ib          [PENT,MMX]

PSRLQ mmxreg,r/m64            ; 0F D3 /r             [PENT,MMX]
PSRLQ mmxreg,imm8             ; 0F 73 /2 ib          [PENT,MMX]
```

`PSxxQ` perform simple bit shifts on the 64-bit MMX registers: the destination (first) operand is shifted left or right by the number of bits given in the source (second) operand, and the vacated bits are filled in with zeros (for a logical shift) or copies of the original sign bit (for an arithmetic right shift).

`PSxxW` and `PSxxD` perform packed bit shifts: the destination operand is treated as a vector of four words or two doublewords, and each element is shifted individually, so bits shifted out of one element do not interfere with empty bits coming into the next.

`PSLLx` and `PSRLx` perform logical shifts: the vacated bits at one end of the shifted number are filled with zeros. `PSRAx` performs an arithmetic right shift: the vacated bits at the top of the shifted number are filled with copies of the original top (sign) bit.

A.131 `PSUBxx`: MMX Packed Subtraction

```
PSUBB mmxreg,r/m64            ; 0F F8 /r             [PENT,MMX]
PSUBW mmxreg,r/m64            ; 0F F9 /r             [PENT,MMX]
PSUBD mmxreg,r/m64            ; 0F FA /r             [PENT,MMX]

PSUBSB mmxreg,r/m64           ; 0F E8 /r             [PENT,MMX]
PSUBSW mmxreg,r/m64           ; 0F E9 /r             [PENT,MMX]

PSUBUSB mmxreg,r/m64          ; 0F D8 /r             [PENT,MMX]
PSUBUSW mmxreg,r/m64          ; 0F D9 /r             [PENT,MMX]
```

`PSUBxx` all perform packed subtraction between their two 64-bit operands, storing the result in the destination (first) operand. The `PSUBxB` forms treat the 64-bit operands as vectors of eight bytes, and subtract each byte individually; `PSUBxW` treat the operands as vectors of four words; and `PSUBD` treats its operands as vectors of two doublewords.

In all cases, the elements of the operand on the right are subtracted from the corresponding elements of the operand on the left, not the other way round.

`PSUBSB` and `PSUBSW` perform signed saturation on the sum of each pair of bytes or words: if the result of a subtraction is too large or too small to fit into a signed byte or word result, it is clipped (saturated) to the largest or smallest value which *will* fit. `PSUBUSB` and `PSUBUSW` similarly perform unsigned saturation, clipping to `0FFh` or `0FFFFh` if the result is larger than that.

A.132 PSUBSIW: MMX Packed Subtract with Saturation to Implied Destination

```
PSUBSIW mmxreg,r/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.115).

A.133 PUNPCKxxx: Unpack Data

```
PUNPCKHBW mmxreg,r/m64        ; 0F 68 /r             [PENT,MMX]
PUNPCKHWD mmxreg,r/m64        ; 0F 69 /r             [PENT,MMX]
PUNPCKHDQ mmxreg,r/m64        ; 0F 6A /r             [PENT,MMX]

PUNPCKLBW mmxreg,r/m64        ; 0F 60 /r             [PENT,MMX]
PUNPCKLWD mmxreg,r/m64        ; 0F 61 /r             [PENT,MMX]
PUNPCKLDQ mmxreg,r/m64        ; 0F 62 /r             [PENT,MMX]
```

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, totalling 64 bits, are then interleaved into the destination, alternating elements from the second (source) operand and the first (destination) operand: so the leftmost element in the result always comes from the second operand, and the rightmost from the destination.

PUNPCKxBW works a byte at a time, PUNPCKxWD a word at a time, and PUNPCKxDQ a doubleword at a time.

So, for example, 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.134 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                [286]
PUSH imm16                    ; o16 68 iw            [286]
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.135 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.127).

A.136 PUSHFx: Push Flags Register

```
PUSHF                         ; 9C                   [186]
PUSHFD                        ; o32 9C               [386]
PUSHFW                        ; o16 9C               [186]
```

PUSHFW 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). PUSHFD pops a doubleword and stores it in the entire flags register.

PUSHF is an alias mnemonic for either PUSHFW or PUSHFD, depending on the current BITS setting.

See also POPF (section A.128).

A.137 PXOR: MMX Bitwise XOR

```
PXOR mmxreg,r/m64             ; 0F EF /r             [PENT,MMX]
```

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.138 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              [286]
RCL r/m16,1                   ; o16 D1 /2             [8086]
RCL r/m16,CL                  ; o16 D3 /2             [8086]
RCL r/m16,imm8                ; o16 C1 /2 ib          [286]
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              [286]
RCR r/m16,1                   ; o16 D1 /3             [8086]
RCR r/m16,CL                  ; o16 D3 /3             [8086]
RCR r/m16,imm8                ; o16 C1 /3 ib          [286]
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 RCR 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 (286 and upwards, beginning with a C1 byte) form of RCL foo,1 by using a BYTE prefix: RCL foo,BYTE 1. Similarly with RCR.

A.139 RDMSR: Read Model-Specific Registers

```
RDMSR                         ; 0F 32                 [PENT]
```

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.165).

A.140 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.

A.141 RDTSC: Read Time-Stamp Counter

```
RDTSC                         ; 0F 31                 [PENT]
```

RDTSC reads the processor's time-stamp counter into EDX:EAX.

A.142 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.143 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             [286]
ROL r/m16,1                   ; o16 D1 /0            [8086]
ROL r/m16,CL                  ; o16 D3 /0            [8086]
ROL r/m16,imm8                ; o16 C1 /0 ib         [286]
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             [286]
ROR r/m16,1                   ; o16 D1 /1            [8086]
ROR r/m16,CL                  ; o16 D3 /1            [8086]
ROR r/m16,imm8                ; o16 C1 /1 ib         [286]
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 ROR 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 3, 4 or 5 bits (depending on the source operand size) of the rotation count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a C1 byte) form of ROL foo,1 by using a BYTE prefix: ROL foo,BYTE 1. Similarly with ROR.

A.144 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.145 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. See also LAHF (section A.90).

A.146 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             [286]
SAL r/m16,1                   ; o16 D1 /4            [8086]
SAL r/m16,CL                  ; o16 D3 /4            [8086]
SAL r/m16,imm8                ; o16 C1 /4 ib         [286]
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 /0                [8086]
SAR r/m8,CL                   ; D2 /0                [8086]
SAR r/m8,imm8                 ; C0 /0 ib             [286]
SAR r/m16,1                   ; o16 D1 /0            [8086]
SAR r/m16,CL                  ; o16 D3 /0            [8086]
SAR r/m16,imm8                ; o16 C1 /0 ib         [286]
SAR r/m32,1                   ; o32 D1 /0            [386]
SAR r/m32,CL                  ; o32 D3 /0            [386]
SAR r/m32,imm8                ; o32 C1 /0 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.152). 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 3, 4 or 5 bits (depending on the source operand size) of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SAL foo,1` by using a `BYTE` prefix: `SAL foo,BYTE 1`. Similarly with `SAR`.

A.147 SALC: Set AL from Carry Flag

```
SALC                          ; D6                   [8086,UNDOC]
```

`SALC` is an early undocumented instruction similar in concept to `SETcc` (section A.150). Its function is to set `AL` to zero if the carry flag is clear, or to `0xFF` if it is set.

A.148 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         [8086]

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. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

To subtract one number from another without also subtracting the contents of the carry flag, use SUB (section A.159).

A.149 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 byte is found.

A.150 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.151 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) into that area as a 32-bit linear address and a 16-bit size limit from that area (in that order). These 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.95).

A.152 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               [286]
SHL r/m16,1                    ; o16 D1 /4              [8086]
SHL r/m16,CL                  ; o16 D3 /4            [8086]
SHL r/m16,imm8                ; o16 C1 /4 ib         [286]
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             [286]
SHR r/m16,1                   ; o16 D1 /5            [8086]
SHR r/m16,CL                  ; o16 D3 /5            [8086]
SHR r/m16,imm8                ; o16 C1 /5 ib         [286]
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.146). 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 3, 4 or 5 bits (depending on the source operand size) of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SHL foo,1` by using a `BYTE` prefix: `SHL foo,BYTE 1`. Similarly with `SHR`.

A.153 `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 5 bits of the shift count are considered.

A.154 `SMI`: System Management Interrupt

```
SMI                           ; F1                   [386,UNDOC]
```

This is an opcode apparently supported by some AMD processors (which is why it can generate the same opcode as INT1), and places the machine into system-management mode, a special debugging mode.

A.155 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.96).

A.156 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.15). To invert the carry flag, use CMC (section A.16).

A.157 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], 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 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.

A.158 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.

A.159 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.148).

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. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

A.160 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 /7 ib              [8086]
TEST r/m16,imm16              ; o16 F7 /7 iw          [8086]
TEST r/m32,imm32              ; o32 F7 /7 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.

A.161 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.

A.162 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. VERW sets the zero flag if the segment can be written.

A.163 WAIT: Wait for Floating-Point Processor

```
WAIT                          ; 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.164 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.84).

A.165 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.139).

A.166 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.167 XBTS: Extract Bit String

```
XBTS reg16,r/m16              ; o16 0F A6 /r           [386,UNDOC]
XBTS reg32,r/m32              ; o32 0F A6 /r           [386,UNDOC]
```

No clear documentation seems to be available for this instruction: the best I've been able to find reads 'Takes a string of bits from the first operand and puts them in the second operand'. It is present only in early 386 processors, and conflicts with the opcodes for CMPXCHG486. NASM supports it only for completeness. Its counterpart is IBTS (see section A.75).

A.168 XCHG: Exchange

```
XCHG reg8,r/m8                ; 86 /r                  [8086]
XCHG reg16,r/m8               ; 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.109).

A.169 XLATB: Translate Byte in Lookup Table

```
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.

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).

A.170 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. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction PXOR (see section A.137) performs the same operation on the 64-bit MMX registers.