B.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 B.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 B.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 B.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 B.2.3; register values are given in Section B.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.

B.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.)

B.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.

B.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).

Prev
Intel x86 Instruction Reference

Home
Up

Next
Key to Instruction Flags