Lecture Notes on Assembly Language - J. Vaughan

13. Instructions and data

From the NASM Manual:

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]

• 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` .

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.2.5.
• 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.2.5; 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.

Data Movement

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.

Arithmetic

B.4.3 ADD: Add Integers

```
ADD r/m8,reg8                ; 00 /r                [8086]
ADD r/m16,reg16              ; o16 01 /r            [8086]
ADD r/m32,reg32              ; o32 01 /r            [386]
ADD reg8,r/m8                ; 02 /r                [8086]
ADD reg16,r/m16              ; o16 03 /r            [8086]
ADD reg32,r/m32              ; o32 03 /r            [386]
ADD r/m8,imm8                ; 80 /7 ib             [8086]
ADD r/m16,imm16              ; o16 81 /7 iw         [8086]
ADD r/m32,imm32              ; o32 81 /7 id         [386]
ADD r/m16,imm8               ; o16 83 /7 ib         [8086]
ADD r/m32,imm8               ; o32 83 /7 ib         [386]
ADD AL,imm8                  ; 04 ib                [8086]
ADD AX,imm16                 ; o16 05 iw            [8086]
ADD EAX,imm32                ; o32 05 id            [386]
```

ADD performs integer addition: it adds its two operands together, and leaves the result in its destination (first) operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.
The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent ADC instruction.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

ADC: Add with Carry

```
ADC r/m8,reg8                   ; 10 /r                 [8086]
ADC r/m16,reg16                 ; o16 11 /r             [8086]
ADC r/m32,reg32                 ; o32 11 /r             [386]
ADC reg8,r/m8                   ; 12 /r                 [8086]
ADC reg16,r/m16                 ; o16 13 /r             [8086]
ADC reg32,r/m32                 ; o32 13 /r             [386]
ADC r/m8,imm8                   ; 80 /2 ib              [8086]
ADC r/m16,imm16                 ; o16 81 /2 iw          [8086]
ADC r/m32,imm32                 ; o32 81 /2 id          [386]
ADC r/m16,imm8                  ; o16 83 /2 ib          [8086]
ADC r/m32,imm8                  ; o32 83 /2 ib          [386]
ADC AL,imm8                     ; 14 ib                 [8086]
ADC AX,imm16                    ; o16 15 iw             [8086]
ADC EAX,imm32                   ; o32 15 id             [386]
```

ADC performs integer addition: it adds its two operands together, plus the value of the carry flag, and leaves the result in its destination (first) operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.
The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent ADC instruction.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. 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.