Intel 80386 Instruction Set Highlights

Peter C. Chapin (Last Revised: January 2, 2017)

Introduction

Although this document talks about the 80386 processor, the instructions presented here work for any processor in the IA-32 architecture family. More modern IA-32 processors typically have additional instructions but they are typically instructions used for specialized purposes and are less useful in a general purpose application context.

In the following description I will use *reg* to refer to one of the 80386 registers and *mem* to refer to a memory location accessible via one of the 80386 addressing modes. For the most part, any instruction that uses *reg* can use any reg, and any instruction that uses *mem* can use any addressing mode. Exceptions are noted.

Be aware that assembly language is *not* standardized. Not only do different processors have (often radically) different instructions but even different assemblers targeting the same processor might use different instruction syntax. The syntax presented here is the "Intel" syntax used by Microsoft's Macro Assembler (MASM) and any assembler that attempts to be compatible with MASM such as Open Watcom's WASM. It is the dominant syntax for assembly language on Windows systems. The other major syntax in use is the "AT&T" syntax used by the GNU Assembler (gas). This alternate syntax dominates on Linux systems where the GNU tool chain is the *de facto* standard.

Addressing Modes

The 80386 supports the following addressing modes. The addressing modes are described by demonstration.

Immediate

The number specified with the instruction is placed into the indicated destination. Note that all three of the forms below do this—including the one with the brackets. Instructions that can use this mode are shown with an *immed* in the instruction summaries below.

```
ANUMBER     EQU         0FFFFFFFFh

            mov         eax, ANUMBER
            mov         ebx, 12345678h
            mov         ecx, [12345678h]
```

Direct Memory

The contents of the specified memory location is placed into the indicated destination. Note that brackets have to used with a segment name (or register) to force MASM to use this addressing mode when a raw address is specified. However, normally this addressing mode is used to access named variables declared in the data segment and no brackets are necessary in that case.

The assembler is sensitive to data types. The three basic types are bytes (8 bits, declared with DB), words (16 bits, declared with DW), and double words (32 bits, declared with DD). Modern processors also deal with quad words (64 bits, declared with DQ). The assembler will select the appropriate instruction variant based on the type of the operand or produce an error message.

```
            .DATA
variable1   DD          12345678h
variable2   DW          1234h
variable3   DB          12h

            .CODE
            mov         eax, variable1
            mov         ebx, ds:[12345678h]     ; Reads specified location.
            mov         cx, variable1           ; Error: variable is a DWORD.
            inc         variable1               ; 32 bit increment used.
            inc         variable2               ; 16 bit increment used.
            inc         variable3               ;  8 bit increment used.
```

Indirect Memory

The contents of some register (often ebx, esi, or edi, but it could be any of the general 32 bit registers) are used as an offset into the data segment. This is the basic mode used when accessing memory with a pointer. The idea is to store the pointer into a 32 bit register and use it to access the operand indirectly.

In cases where the data type must be specified the notation BYTE PTR, WORD PTR, or DWORD PTR must be used. The assembler will produce an ambiguity error if disambiguation is needed.

```
            mov         eax, [ebx]        ; Read 32 bits from memory.
            mov         cl, [esi]         ; Read  8 bits from memory.
            inc         [edi]             ; Ambiguous. What size is intended?
            inc         DWORD PTR [edi]   ; 32 bit increment used.
            inc         WORD  PTR [edi]   ; 16 bit increment used.
            inc         BYTE  PTR [edi]   ;  8 bit increment used.
```

Indirect Memory with (Scaled) Index and Optional Displacement.

More generally the indirect mode can use the following form.

```
[base_reg + scale*index_reg + displacement]
```

where

- "base_reg" is one of the general purpose registers (eax, ebx, ecx, edx, esi, edi, ebp, or esp). When ebp or esp is used as a base, the reference is into the stack segment.
- "scale" is either 1, 2, 4, or 8. A scale factor of 1 is the default and need not be mentioned explicitly.

- “index_reg” is one of the general purpose registers (eax, ebx, ecx, edx, esi, edi, or ebp). Notice that esp can not be used as an index register.
- “displacement” is a *signed* quantity that is either 8 or 32 bits (if necessary) in size.

This general mode is particularly useful in connection with arrays which are just allocated blocks of memory with a label representing their starting address.

```
            .DATA
buffer1     DB          1000 dup(0)       ; Array of 1000 bytes.

            mov         al, [ebx+2]       ; 8 bits from location ebx + 2.
            mov         al, ebx[2]        ; Alternate form of above.
            mov         al, [ebx+buffer1] ; 8 bits from the ebxth element.
            mov         al, buffer1[ebx]  ; Alternate form of above.
            Mov         eax, [ebx+4*esi]  ; 32 bits from the esith element.
```

In general to access an array element one has to multiply the index value by the size of the element type and add the result to the base address of the array. The addressing modes of the 80386 make this easy for arrays of bytes, words, or double words. For larger objects it is necessary to use explicit multiply instructions.

Note that the ability to add a displacement allows you to easily access the components of structures in an array of structures. Adjust the index register to “point” at the array element in question and then use the displacement for dealing with the offset into the structure at that position.

Use of the "Other" Base Register

The ebp register can be as a base register. When the ebp register is used, however, the offset calculated is taken to be an offset into the stack segment. This allows easy access of data on the stack. Typically programming languages (such as C) use the stack to pass parameters to functions and for function local variables.

The Instruction Set

1. Data movement. Despite the name, “mov” these instructions actually copy data. The source of the data is not affected. The source operand is the rightmost (last) operand. The destination operand is the leftmost (first) operand.
   a) `mov   reg, reg`
   b) `mov   mem, reg`
   c) `mov   reg, mem`
   d) `mov   mem, immed`
   e) `mov   reg, immed`
2. Math and bitwise operations.

a) `add   reg, reg`

b) `add   mem, reg`

c) `add   reg, mem`

d) `add   reg, immed`

e) `add   mem, immed`

The same pattern is used for "adc" (add with carry), "sub" (subtract), "sbb" (subtract with borrow), "and" (bitwise AND), "or" (bitwise OR), and "xor" (bitwise XOR). Note that the instruction "sub ebx, ecx" subtracts ecx from ebx and leaves the result in ebx. That is, the instruction calculates ebx -= ecx.

3. Compare and jump

a) The cmp instruction works like the math and bitwise instructions above. It does a subtraction but does not store result. However, it does condition the flags in preparation for a jump.

b) The test instruction works like the math and bitwise instructions above. It does a bitwise "and" but does not store result. However, it does condition the flags in preparation for a jump.

c) Unsigned jumps

• `jb`	`Jump if below.`
• `jbe`	`Jump if below or equal.`
• `je`	`Jump if equal.`
• `jae`	`Jump if above or equal.`
• `ja`	`Jump if above.`

d) Signed jumps

• `jl`	`Jump if less.`
• `jle`	`Jump if less or equal.`
• `je`	`Jump if equal.`
• `jge`	`Jump if greater or equal.`
• `jg`	`Jump if greater.`

e) Jump on state of individual flags

• `js`	`Jump if sign flag set.`
• `jc`	`Jump if carry flag set.`
• `jz`	`Jump if zero flag set.`
• `jo`	`Jump if overflow flag set.`
• `jp`	`Jump if parity flag set (even parity)`

f) An "n" can be used in all forms to negate the sense of the jump. For example

- `jnb        Jump if not below (unsigned jump).`
- `jnbe       Jump if not below or equal.`
- `jne        Jump if not equal.`
- `jnae       Jump if not above or equal.`
- `jna        Jump if not above.`
- ...etc

4. Shift and rotate
For both the shift and rotate instructions the number of bits shifted (rotated) can be either 1 or the number in cl. A reg or mem can be shifted (rotated).

a) Shifts. An arithmetic right shift copies the sign flag. A logical right shift does zero fill.

- `sal        ax,1       ; Shift arithmetic left.`
- `sar        ax,1       ; Shift arithmetic right.`
- `shl        ax,1       ; Shift (logical) left.`
- `shr        ax,1       ; Shift (logical) right.`

b) Rotates

- `rol        ax,1       ; Rotate left.`
- `ror        ax,1       ; Rotate right.`
- `rcl        ax,1       ; Rotate left through carry.`
- `rcr        ax,1       ; Rotate right through carry.`

5. Direct Flag Control

a) `stc        ; Set carry flag.`

b) `std        ; Set direction flag (string instr go down).`

c) `sti        ; Enable interrupts.`

d) `clc        ; Clear carry flag.`

e) `cld        ; Clear direction flag (string instr go up).`

f) `cli        ; Disables interrupts.`

g) `cmc        ; Complements the carry flag.`

Special Instructions

The 8086 provides a number of specialized instructions that can be used to improve efficiency. Compilers often don't use these instructions effectively because to do so requires extensive analysis of the program. Human programmers, however, try to use these features as much as they can.

The loop instruction provides a simplified way to set up a loop. It uses ecx as a counter register. The instruction decrements ecx and then jumps to the indicated address if ecx is not zero.

```
                mov   ecx, 1000         ; 1000 times through the loop.
top:            ...
                ...
                ...
                loop  top               ; Jump to top if more to do.
```

The movsb (movsw, movsd) instruction moves a byte (word, double word)from ds:esi to es:edi (note the use of the extra segment). The esi and edi registers are automatically incremented or decremented (depending on the state of the direction flag). These instructions are often used with a "rep" prefix to force them to run ecx number of times. The overall effect is to implement a block copy instruction.

```
          mov         ecx, 1000         ; 1000 words to move.
          mov         esi, OFFSET source
          mov         edi, OFFSET destin
          rep movsw                     ; Copy all 1000 words.
```

The stosb (stosw, stosd) and lodsb (lodsw, lodsb) store (load) a byte (word, double word) between memory and the accumulator. These instructions use esi and edi in the same way movsX instructions do. They can also be used with a "rep" prefix.

```
          mov         ecx, 1000         ; 1000 bytes to store.
          mov         edi, OFFSET destin
          mov         al, 0aah
          rep stosb                     ; Copy al into memory 1000 times.
```

This description is not a complete description of all the 80386 instructions. However, this list should suffice for the comprehension of many programs.