X86 Assembly/Machine Language Conversion

Contents

Relationship to Machine Code

X86 assembly instructions have a one-to-one relationship with the underlying machine instructions. This means that essentially we can convert assembly instructions into machine instructions with a look-up table. This page will talk about some of the conversions from assembly language to machine language.

CISC and RISC

The x86 architecture is a **complex instruction set computer** (CISC) architecture. Amongst other things, this means that the instructions for the x86 architecture are of varying lengths. This can make the processes of assembly, disassembly and instruction decoding more complicated, because the instruction length needs to be calculated for each instruction.

x86 instructions can be anywhere between 1 and 15 bytes long. The length is defined separately for each instruction, depending on the available modes of operation of the instruction, the number of required operands and more.

8086 instruction format (16 bit)

This is the general instruction form for the 8086 sequentially in main memory:

Prefixes (optional)							
Opcode (first byte)					D	W	
Opcode 2 (occasional second byte)							
MOD			Reg			R/M	
Displacement or data (occasional: 1, 2 or 4 bytes)							

Prefixes
: Optional prefixes which change the operation of the instruction

D
: (1 bit) Direction. 1 = Register is Destination, 0 = Register is source.

W
: (1 bit) Operation size. 1 = Word, 0 = byte.

Opcode

the opcode is a 6 bit quantity that determines what instruction family the code is

MOD (Mod)

(2 bits) Register mode.

Reg

(3 bits) Register. Each register has an identifier.

R/M (r/m)

(3 bits) Register/Memory operand

Not all instructions have W or D bits; in some cases, the width of the operation is either irrelevant or implicit, and for other operations the data direction is irrelevant.

Notice that Intel instruction format is little-endian, which means that the lowest-significance bytes are closest to absolute address 0. Thus, words are stored low-byte first; the value 1234H is stored in memory as 34H 12H. By convention, most-significant bits are always shown to the left within the byte, so 34H would be 00110100B.

After the initial 2 bytes, each instruction can have many additional addressing/immediate data bytes.

Mod / Reg / R/M tables

Mod	Displacement
00	If r/m is 110, Displacement (16 bits) is address; otherwise, no displacement
01	Eight-bit displacement, sign-extended to 16 bits
10	16-bit displacement (example: MOV [BX + SI]+ displacement,al)
11	r/m is treated as a second "reg" field

Reg	W = 0	W = 1	double word
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

r/m	Operand address
000	(BX) + (SI) + displacement (0, 1 or 2 bytes long)
001	(BX) + (DI) + displacement (0, 1 or 2 bytes long)
010	(BP) + (SI) + displacement (0, 1 or 2 bytes long)
011	(BP) + (DI) + displacement (0, 1 or 2 bytes long)
100	(SI) + displacement (0, 1 or 2 bytes long)
101	(DI) + displacement (0, 1 or 2 bytes long)
110	(BP) + displacement unless mod = 00 (see mod table)
111	(BX) + displacement (0, 1 or 2 bytes long)

Note the special meaning of MOD 00, r/m 110. Normally, this would be expected to be the operand [BP]. However, instead the 16-bit displacement is treated as the absolute address. To encode the value [BP], you would use mod = 01, r/m = 110, 8-bit displacement = 0.

Example: Absolute addressing

Let's translate the following instruction into machine code:

```
XOR CL, [12H]
```

Note that this is XORing CL with the contents of address 12H – the square brackets are a common indirection indicator. The opcode for XOR is "001100dw". D is 1 because the CL register is the destination. W is 0 because we have a byte of data. Our first byte therefore is "00110010".

Now, we know that the code for CL is 001. Reg thus has the value 001. The address is specified as a simple displacement, so the MOD value is 00 and the R/M is 110. Byte 2 is thus (00 001 110b).

Byte 3 and 4 contain the effective address, low-order byte first, 0012H as 12H 00H, or (00010010b) (00000000b)

All together,

```
XOR CL, [12H] = 00110010 00001110 00010010 00000000 = 32H 0EH 12H 00H
```

Example: Immediate operand

Now, if we were to want to use an immediate operand, as follows:

```
XOR CL, 12H
```

In this case, because there are no square brackets, 12H is immediate: it is the number we are going to XOR against. The opcode for an immediate XOR is 1000000w; in this case, we are using a byte, so w is 0. So our first byte is (10000000b).

The second byte, for an immediate operation, takes the form "mod 110 r/m". Since the destination is a register, mod is 11, making the r/m field a register value. We already know that the register value for CL is 001, so our second byte is (11110 001b).

The third byte (and fourth byte, if this were a word operation) are the immediate data. As it is a byte, there is only one byte of data, 12H = (00010010b).

All together, then:

```
XOR CL, 12H = 10000000 11110001 00010010 = 80H F1H 12H
```

x86-32 Instructions (32 bit)

The 32-bit instructions are encoded in a very similar way to the 16-bit instructions, except (by default) they act upon dword quantities rather than words. Also, they support a much more flexible memory addressing format, which is made possible by the addition of an SIB "scale-index-base" byte, which follows the ModR/M byte.

x86-64 Instructions (64 bit)