

# Introduction to Intel Machine Language (and More On Linking)

Norman Matloff

March 18, 2007

© 2002-2007, N.S. Matloff

## Contents

<b>1 Overview</b>	<b>2</b>
<b>2 Relation of Assembly Language to Machine Language</b>	<b>2</b>
<b>3 Example Program</b>	<b>3</b>
3.1 The Code . . . . .	3
3.2 Feedback from the Assembler . . . . .	4
3.3 A Few Instruction Formats . . . . .	4
3.4 Format and Operation of Jump Instructions . . . . .	5
3.5 Other Issues . . . . .	6
<b>4 It Really Is Just a Mechanical Process</b>	<b>7</b>
<b>5 You Could Write an Assembler!</b>	<b>7</b>
<b>6 A Look at the Final Product</b>	<b>7</b>

# 1 Overview

This document serves as a brief introduction to Intel machine language. It assumes some familiarity with Intel assembly language (using the AT&T syntax).

## 2 Relation of Assembly Language to Machine Language

Assembly language (AL) is essentially identical to machine language (ML). The only difference is that ML expresses any machine instruction using the 1s and 0s on which the machine operates, while AL expresses the same instruction using English-like abbreviations for specific bit patterns.

Consider for instance a register-to-register “move” operation, which copies the contents of one register to another.<sup>1</sup> The bit pattern which signals this, called an **op code**, is 1000100111. The circuitry is set up so that when the CPU sees 1000100111 in the first ten bits of an instruction, it knows it must do a register-to-register move.<sup>2</sup>

As humans we get tired of writing long strings of 1s and 0s, and it would also be burdensome to have to remember which ones signify which op codes. As an alternative, we could hire a clerk. We would say to the clerk, “Clerk, every time you see me write **movl**, you enter 1000100111.” The name **movl** is a lot easier for us to remember than 1000100111, so would be a great convenience for us (though the poor clerk would have to deal with these things).

The circuitry in the CPU is also set up to know that register-to-register “move” instructions, i.e. instructions whose first bits are 1000100111, are two bytes long.<sup>3</sup> Also, the circuitry is set up to know that the source and destination operands of the move—which register is to be copied to which other register—are stored in the remaining six bits in this instruction: The eleventh, twelfth and thirteenth bits specify the source register, and the fourteenth, fifteenth and sixteenth indicate the destination register, according to the following codes:

EAX	000
EBX	011
ECX	001
EDX	010

(The Intel CPU also has a number of other registers, but we will not list their codes here.)

Again, instead of having to remember the code for say, EBX, we would tell the clerk “Whenever I say `%ebx`, I mean 011,” and so on. Well, our “clerk” is the assembler, e.g. **as** in standard Unix systems. We use a text editor to type those same abbreviations into a file, and the assembler converts them to 1s and 0s just as a clerk would.

Remember, though, that we the programmer are still in full control of which instructions (i.e. which operations and operands) to specify, in contrast to a compiler. If for example we type the line

```
movl %eax, %ebx
```

---

<sup>1</sup> The term “move” is thus a misnomer, though by now entrenched in computer jargon.

<sup>2</sup> What do we mean by the “first” 10 bits? An Intel instruction can be several bytes in length. When we speak of the “first” byte of an instruction, we mean the one contained in the lowest-numbered memory address. When we speak of the “first” bit within a byte, we mean the most-significant bit within that byte.

<sup>3</sup>Note that that means that the circuitry will advance the Program Counter register by 2, so as to prepare for the fetch of the instruction which follows the move instruction.

into a file, say `x.s`, we know the assembler will convert it to the machine instruction which does this operation, which we will see below happens to be 1000100111000011, i.e. 0x89c3. By contrast, if we run

```
y = x + 3;
```

through a compiler, we have no idea what machine instructions the compiler will generate from it. Indeed, different compilers would likely generate different machine instructions.

The assembler places the machine code it generates, e.g. 1000100111000011 above, into the object file, `x.o`.

## 3 Example Program

### 3.1 The Code

We ran an assembly language source file, `Totals.s`, through the `as` assembler with the `-a` option, the latter requesting the assembler to print out for us the machine code it produces, side-by-side with our original assembly language. Here is the output:

```
GAS LISTING Total.s page 1
```

```
1
2         # finds the sum of the elements of an array
3
4         # assembly/linking Instructions:
5
6         # as -a --gstabs -o total.o Total.s
7         # ld -o total total.o
8
9         .data # start of data section
10
11        x:
12 0000 01000000    .long 1
13 0004 05000000    .long 5
14 0008 02000000    .long 2
15 000c 12000000    .long 18
16
17        sum:
18 0010 00000000    .long 0
19
20        .text # start of code section
21
22        .globl _start
23        _start:
24 0000 B8040000    movl $4, %eax # EAX will serve as a counter for
25                # number of words left
26 0005 BB000000    movl $0, %ebx # EBX will store the sum
27 000a B9000000    movl $x, %ecx # ECX will point to the current
28                # element to be summed
29 000f 0319    top:  addl (%ecx), %ebx
30 0011 83C104    addl $4, %ecx # move pointer to next element
31 0014 48        decl %eax     # decrement counter
32 0015 75F8        jnz top      # if counter not 0, loop again
33 0017 891D1000 done: movl %ebx, sum # done, store result in "sum"
34 001d 8D7600
```

```

DEFINED SYMBOLS
    Total.s:11      .data:00000000 x
    Total.s:17      .data:00000010 sum
    Total.s:23      .text:00000000 _start
    Total.s:29      .text:0000000f top
    Total.s:33      .text:00000017 done

```

```
NO UNDEFINED SYMBOLS
```

## 3.2 Feedback from the Assembler

Each line in this output is organized as follows:

Display Line Number	Memory Offset	Contents	Assembly Source Line
---------------------	---------------	----------	----------------------

The **offset** is a relative memory address, i.e. the distance from the given item to the beginning of its section (**.data**, **.text**, etc.).

Here the clerk is saying to us, “Boss, here is what I did with that assembly-language source line of yours, which I’ve shown here in the fourth field. I translated it to the coding you see in the third field, and I arranged for this coding to go into the offset I’ve listed in the second field. For your convenience, I’ve also shown the line number from your source file in the first field.”

What does the word *offset* mean? For instance, look at Line 18. It says that the word which we labeled **sum** in our source code will be located  $0x10 = 16$  bytes from the beginning of the **.data** section.

## 3.3 A Few Instruction Formats

The Contents field shows us the bytes produced by the assembler from our source code, *displayed byte-by-byte, in order of increasing address*. Note the effect of little-endian-ness, for instance in Line 13. That line says that our assembly code

```
.long 5
```

produced the sequence of bytes 05 00 00 00. That makes sense, right? In the number 00000005, the least significant byte is 05 (recall that a byte is two hex digits), so it will have the lowest address, and thus it will be displayed first.

The actual address of this word will be determined when the program is linked by **ld**. The latter will choose a place in memory at which to begin the **.data** section, and that will determine everything else goes. If that section begins at, say,  $0x2020$ , then for instance the data corresponding to Line 15 will be at case  $0x2020+0x0c = 0x202c$ .

Note that the addresses assigned to these items in the **.data** section of our source file are contiguous and follow the same order in which we “declared” them in the source file.

Now, let’s learn a few more Intel machine-language formats, to add to the register-to-register move we saw above. First, let’s consider the one on Line 24.

This instruction is an immediate-to-register move, where the word “immediate” means “the source operand is a constant which appears in the instruction.” Here the constant is 4; it is a 32-bit integer occupying the last four bytes of this five-byte instruction.

This type of instruction is coded by its first five bits being 10111. The next three bits code the destination register, in this case being 000. So the first byte of the instruction is 10111000, i.e. 0xb8, followed by the constant 04000000 (again, note the effect of Intel CPUs being little-endian). We will describe the format for immediate-to-register move as 10111DDDIMM4, where 10111 is the op code, DDD denotes the destination register and IMM4 denotes a 4-byte immediate constant.<sup>4</sup>

Note also on Line 24 that the offset-field count has been reset back to 0000 again, since we are now in a new section (**.text**).

Lines 26 and 27 are similar, but one aspect must be noted in the latter. The immediate constant here, **\$x**, is the address of **x**. But all the assembler knows at this point is that later, when the program is linked, **x** will be 0 bytes from the start of the **.data** section. If **ld** allocates the **.data** section at, say, 0x2000, then **x** will be at that address. So, ideally the immediate constant placed by the assembler into the machine instruction here would be 0x2000 (or wherever else the **.data** section ends up). But, since this value is unknown at assembly time, the assembler just puts the offset, 0, in the instruction on a temporary basis, to be resolved later by the linker. ✓

In Line 29 we see an indirect-to-register add. Its op code is 0000001100, followed by three bits for the destination register and then three bits for the source register (the latter being used in indirect mode). We will state this notationally as 0000001100DDDSSS. An add which is register-to-indirect, e.g. **addl %ecx, (%ebx)**, has the format 0000000100SSSDDD.

Of course, the choice of the codes for operations, registers and addressing modes is arbitrary, decided by the hardware designers. Typically it is chosen according to two considerations:

- Instructions which are expected to be more commonly used should have shorter codes, in order to conserve memory and disk space.
- The circuitry may be easier to design, or made faster, with some codes.

In Line 31 we see a register decrement operation. A little experimentation—changing the register to EBX and then observing how the machine code changes—reveals that the format for this kind of instruction is 01001DDD, where DDD is the destination register.

### 3.4 Format and Operation of Jump Instructions

There is quite a bit to discuss on Line 32. The format for this jump-if-Zero-flag-not-set instruction is 01110101IMM1, where 01110101 is the op code and IMM1 is a 1-byte immediate signed constant which indicates the **target** of the jump, i.e. where to jump to if the Zero flag is not set. The constant is actually the (signed) distance to the target, as follows.

Recall that any computer has a register whose generic name is the Program Counter (PC) which points to the next instruction to be executed. Let's say that the **.text** part of the program starts at 0x300ba. Then the instruction shown here in Line 32 will be at 0x300ba+0x15 = 0x300cf, and our target (shown in Line 29) will be at address 0x300ba+0xf = 0x300c9.

Recall also that as soon as an instruction is fetched for execution, the PC, which had been pointing to that instruction, is incremented to point to the next instruction. In the case of the instruction in Line 32, here is how the incrementing occurred: The PC had contained 0x300cf, and the CPU fetched the instruction from

<sup>4</sup>We describe our earlier register-to-register move format as 1000100111SSSDDD.

that location. Then the CPU, seeing that the instruction begins with 01110101, realizes that this is a JNZ instruction, thus 2 bytes in length. So, the CPU increments the PC by 2, to point to the instruction in Line 33, at  $0x300cf+2 = 0x300d1$ .

Now, to execute the JNZ instruction, the CPU checks the Zero flag. If the flag is not set, the CPU adds the IMM1 part of the JNZ instruction to the current PC value, treating IMM1 as an 8-bit 2s complement number. As such,  $0xf8 = -8$ , so the sum in question will be  $0x300d1-8 = 0x300c9$ . This latter quantity is exactly what we want, the address of our target. The CPU puts this number in the PC, which means that the next instruction fetch will be done from location  $0x300c9$ —in other words, we jump to the line we had named **top** in our assembly language source file, just as desired. ✓

Of course, the assembler knows that the CPU will add IMM1 to the current PC value, and thus sets IMM1 accordingly. Specifically, the assembler computes IMM1 to equal the distance from the instruction following the JNZ instruction to the target instruction.<sup>5</sup>

IMM1 is just 8 bits. If the distance to the target is greater than what can be expressed as an 8-bit 2s complement number, the Intel instruction set does have an alternate version of JNZ with format 000011110000101IMM4.

Keep in mind that the circuitry which implements JNZ will be quite simple. It will check the Zero Flag, and if that flag is 1, the circuitry will then add the second byte of the instruction (-8 in our example here) to the EIP register. That's all!

### 3.5 Other Issues

The assembler goes through the **.s** source file line by line, translating each line to the corresponding machine code. By the time the assembler sees this JNZ instruction, it has already seen the line labeled **top**. Thus it knows the offset of **top** within the **.text** section, and can compute the distance to that target, as we saw here. But what if the jump target were to come after this line with JNZ? In this case, the assembler won't know the offset of the target yet, and thus will not be able to completely translate the instruction yet. Instead, the assembler will make a note to itself, saying that it will come back and finish this line after it finishes its first pass through the **.s** file. So, assemblers are **two-pass**, going through the **.s** file twice.

In doing the computation  $0x300d1-8$ , the CPU must first make the two quantities compatible in terms of size. So, the CPU will do a **sign extension**, extending the 8-bit version of -8, i.e.  $0xf8$ , to the 32-bit version, i.e.  $0xffffffff$ . So, technically, the CPU is performing  $0x000300d1+0xffffffff$ , which does indeed come out to  $0x300c9$ .

The formats for some other conditional jump instructions are: JZ, 01110100IMM1; JS format 01111000IMM1; JNS 01111001IMM1; and so on. The unconditional jump JMP has format 11101011IMM1.

The Intel instruction set contains hundreds of instructions, and their formats become quite complex. We will not pursue this any further.

Note, though, that even in the few formats we've seen here, there is a large variation in instruction length. Some instructions are only one byte long, some are two bytes, and so on, with lengths ranging up to six bytes in the instructions here. This is in great contrast to RISC CPUs such as MIPS, in which all instructions are four bytes long. RISC architectures aim for uniformity of various kinds, which helps make a "lean and mean" machine.

---

<sup>5</sup>This implies that it doesn't matter that the execution-time address of **top** is unknown at assembly time, unlike the case in Line 27, where the lack of knowledge of the execution-time address of **x** caused a problem.

## 4 It Really Is Just a Mechanical Process

As pointed out earlier, the assembler is just acting like a clerk, mechanically translating English-like codes to the proper bits. To illustrate this, note that we could “write” our own machine language in hex, right there in the middle of our assembly code! For instance, recall from our earlier example that the instruction **decl %eax** has the machine language 0x48. We could set that 0x48 ourselves:

```
... # same code as before, but not shown here
    movl $x, %ecx
top:  addl (%ecx), %ebx
    addl $4, %ecx
    .byte 0x48 # we write "directly" in machine language
    jnz top
done: movl %ebx, sum
```

With that **.byte** directive, we are simply saying to the assembler, “After you translate **addl \$4, %ecx** to machine code, put a byte 0x48 right after that code.” Since this is what the assembler would have done anyway in assembling **decl %eax**, we get exactly the same result. Try assembling and running the program with this change; you’ll see that it works fine.

## 5 You Could Write an Assembler!

A common exercise in assembly language courses is to write a simple assembler. Even if you are not assigned to do so, it would be a very enriching exercise if you were to at least give some thought as to how you would do it. For example, just stick to the few instructions covered in this unit. Your job would be to write a program, say in C, which translates assembly code for the instructions and addressing modes covered in this unit to machine language.

## 6 A Look at the Final Product

The linker chooses addresses for the sections. We can determine those, for example, by running

```
% readelf -s tot
```

on our executable file **tot**. The relevant excerpt of the output is

```
 9: 08049094      0 NOTYPE  LOCAL  DEFAULT  2 x
10: 080490a4      0 NOTYPE  LOCAL  DEFAULT  2 sum
11: 08048083      0 NOTYPE  LOCAL  DEFAULT  1 top
12: 0804808b      0 NOTYPE  LOCAL  DEFAULT  1 done
13: 08048074      0 NOTYPE  GLOBAL DEFAULT  1 _start
```

So we see that **ld** has arranged things so that, when our program is loaded into memory at run time, the **.data** section will start at 0x08049094 and the **.text** section at 0x08048074.

We can use GDB to confirm that these addresses really hold at run time:

```
% gdb tot
(gdb) p/x &x
$1 = 0x8049094
(gdb) p/x &_start
$2 = 0x8048074
```

We can also confirm that the linker really did fix the temporary machine code the assembler had produced from

```
movl $x, %ecx
```

Recall that the assembler didn't know the address of `x`, since the location of the `.data` section would not be set until later, when the linker ran. So, the assembler left a note in the `.o` file, asking the linker to put in the real address. Let's check that it did:

```
(gdb) b 24
Breakpoint 1 at 0x804807e: file tot.s, line 24.
(gdb) r
Starting program: /root/50/tot
Breakpoint 1, _start () at tot.s:24
24      movl $x, %ecx # ECX will point to the current
Current language: auto; currently asm
(gdb) disassemble
Dump of assembler code for function _start:
0x08048074 <_start+0>: mov     $0x4,%eax
0x08048079 <_start+5>: mov     $0x0,%ebx
0x0804807e <_start+10>: mov     $0x8049094,%ecx
End of assembler dump.
(gdb) x/5b 0x0804807e
0x0804807e <_start+10>: 0xb9    0x94    0x90    0x04    0x08
```

We see that the machine code for the instruction really does contain the actual address of `x`.

