Introduction to Linux Intel Assembly Language

Norman Matloff

February 5, 2002

Contents

1 Overview

This document introduces the use of assembly language on Linux systems. The intended audience is students in the first week or two of a computer systems/assembly language course. It is assumed that the reader is already familiar with Unix, and has been exposed a bit to the Intel register and instruction set.

2 Different Assemblers

Our emphasis will be on **as** (also written sometimes as **gas**, for ``GNU assembler''), the assembler which is part of the **gcc** package. Its syntax is commonly referred to as the ``AT&T syntax,'' alluding to Unix's Bell Labs origins.

However, we will also be using another commonly-used assembler, NASM. It uses Intel's syntax, which is similar to that of **as** but does differ in some ways. For example, for two-operand instructions, **as** has us specify the source first while NASM wants the destination first.

It is very important to note, though, that the two assemblers will produce the same machine code. Unlike a compiler, whose output is unpredictable, we know ahead of time what machine code an assembler will produce, because the assembly-language *mnemonics* are merely handy abbreviations for specific machine-language bit fields.

Suppose for instance we wish to copy the contents of the AX register to the BX register. In **as** we would write

```
mov %ax,%bx
```

while in NASM it would be

```
MOV BX,AX
```

but the same machine-language will be produced in both cases, 0x6689c3.

3 Assembler Command-Line Syntax

To assemble an AT&T-syntax source file, say x.s (UNIX custom is that assembly-language files end with a .s suffix), we will type

```
as -a --gstabs -o x.o x.s
```

The -o option specifies what to call the *object file*, i.e. machine-code file, which is the primary output of the assembler. The -o means we are telling the assembler, ``The name we want for the .o file immediately follows,'' in this case x.o.

The -a option tells the assembler to display to the screen the source code, machine code and segment offsets side-by-side, for easier correlation.

The -gstabs option tells the assembler to retain in x.o the *symbol table*, a list of the locations of whatever labels are in x.s, in the object file. This is used by symbolic debuggers, in our case **gdb** or **ddd**.

If the file were instead in Intel syntax, our command would be

```
nasm -f elf -o x.o -l x.l x.s
```

The -f option instructs the assembler to set up the x.o file so that the executable file constructed from it later on will be of the ELF format, which is a common executable format on Linux platforms. The -l option plays a similar role to -a in **as**, in that a side-by-side listing of source and machine code will be written to the file x.l.[1]

Things are similar under other operating systems.[2] Using the Microsoft or Turbo compilers, for example, assembly language source files have the suffix .asm, object files have the suffix .obj, etc.

4 Sample Program

In this very simple example, we find the sum of the elements in a 4-word array, x.

First, the program using AT&T syntax:

```
# introductory example; finds the sum of the elements of an array

.data  # start of data segment

x:
      .long   1
      .long   5
      .long   2
      .long   18

sum:
      .long 0

.text  # start of code segment

.globl _start
_start:
      movl $4, %eax  # EAX will serve as a counter for
                     # the number of words left to be summed
      movl $0, %ebx  # EBX will store the sum
      movl $x, %ecx  # ECX will point to the current
                     # element to be summed
top:  addl (%ecx), %ebx
      addl $4, %ecx  # move pointer to next element
      decl %eax  # decrement counter
      jnz top  # if counter not 0, then loop again
done: movl %ebx, sum  # done, store result in "sum"
```

And the version using Intel syntax:

```
; introductory example; finds the sum of the elements of an array

SECTION .data ; start of data segment

global x
x:
      dd      1
      dd      5
      dd      2
      dd      18

sum:
      dd   0

SECTION .text ; start of code segment

      mov  eax,4     ; EAX will serve as a counter for
                    ; the number words left to be summed
      mov  ebx,0     ; EBX will store the sum
      mov  ecx, x    ; ECX will point to the current
                     ; element to be summed
top:  add  ebx, [ecx]
      add  ecx,4     ; move pointer to next element
      dec  eax   ; decrement counter
      jnz top  ; if counter not 0, then loop again
done: mov  [sum],ebx  ; done, store result in "sum"
```

Let's discuss this in the context of the AT&T syntax.

First, we have the line

```
.data   # start of data segment
```

The fact that this begins with `.' signals the assembler that this will be a *directive*, meaning a command to the assembler rather than something the assembler will translate into an instruction. (The `#' character means that it and the remainder of the line are to be treated as a comment.) This directive here is indicating that what follows will be data rather than code.

Next

```
x:

      .long   1
      .long   5
      .long   2
      .long   18
```

This tells the assembler to make a note in x.o saying that when this program is later loaded for execution, there will be four consecutive ``long'' (i.e. 32-bit) words in memory set with initial values 1, 5, 2 and 18 (decimal).[3] Moreover, we are telling the assembler that in our assembly code below, the first of these four long words will be referred to as x. We say that x is a *label* for this word.[4] Similarly, immediately following those four long words in memory will be a long word which we will refer to in our assembly code below as sum.

By the way, what if x had been an array of 1,000 long words instead of four, with all words to be initialized to, say, 8? Would we need 1,000 lines? No, we could do it this way:

```
x:
      .rept 1000
      .long 8
      .endr
```

The .rept directive tells the assembler to act as if the lines following .rept, up to the one just before .endr, are repeated the specified number of times.

Next we have a directive signalling the start of the *text* segment, meaning actual program code. Look at the first two lines:

```
_start:
      movl $4, %eax
```

Here _start is another label, in this case for the location in memory at which execution of the program is to begin, called the **entry point**, in this case that **movl** instruction. We did not choose the name for this label arbitrarily, in contrast to all the others; the UNIX linker takes this as the default.

The **movl** instruction copies the constant 4 to the EAX register.[5] The `l' in ``movl'' means ``long.'' The corresponding Intel syntax,

```
      mov eax,4
```

has no such distinction, relying on the fact that EAX is a 32-bit register to implicitly give the same message to the assembler.

The second instruction is similar, but there is something noteworthy in the third:

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

In the token $4 in the first instruction, the dollar sign meant a constant, and the same is true for $x. The constant here is the address of x. Thus the instruction places the address of x in the ECX register, so that EAX serves as a pointer. A later instruction,

```
      addl $4, %ecx
```

increments that pointer by 4 bytes, i.e. 1 word, each time we go around the loop, so that we eventually have the sum of all the words.

Note that $x has a completely different meaning that x by itself. The instruction

```
      movl x, %ecx
```

would copy the memory location x itself, rather than its address, to ECX.[6]

The next line begins the loop:

```
top:  addl (%ecx), %ebx
```

Here we have another label, ``top'', a name which we've chosen to remind us that this is the top of the loop. This instruction takes the word pointed to by ECX and adds it to EBX. The latter is where I am keeping the total.

Recall that eventually we will copy the final sum to the memory location labeled ``sum''. We don't want to do so within the loop, though, because memory access is slow and we thus want to avoid it. So, we keep our sum in a register, and copy to memory only when we are done.[7]

If we were not worried about memory access speed, we might directly to the variable ``sum'', as follows:

```
      movl $sum,%edx  # use %edx as a pointer to "sum"
      movl $0,%ebx
top:  addl (%ecx), %ebx  # old sum is still in %ebx
      movl %ebx,(%edx)
```

We could NOT do

```
      movl $sum,%edx  # use %edx as a pointer to "sum"
top:  addl (%ecx),(%edx)
```

because Intel chips (like most CPUs) do not allow an instruction to have both its source and destination operands in memory.[8]

The bottom part of the loop is:

```
        decl %eax
        jnz top
```

The **decl** (``decrement long'') instruction subtracts 1 from EAX. The hardware also records whether the result of this instruction is 0, in the Zero Flag in the CPU. The **jnz** instruction says, ``If the result of the last arithmetic operation was not 0, then jump to the instruction labeled ``top''.'' So, the net effect is that we will go around the loop four times, until EAX reaches 0, then exit the loop (where ``exiting'' the loop merely means going to the next instruction, rather than jumping to the line labeled ``top'').

Note that the label ``done'' was my choice, not a requirement of **as**, and I didn't need a label for that line at all, since it is not referenced elsewhere in the program. I included it only for the purpose of debugging, as seen later.

5 16-Bit, 8-Bit and String Operations

Recall the following instruction from our example above:

```
addl (%ecx), %ebx
```

How would this change if we had been storing our numbers in 16-bit memory chunks?

In order to do that, we would use .word instead of .long for initialization in the .data segment. The above instruction would become

```
add (%ecx), %bx
```

The changes here are self-explanatory, but the non-change may seem odd at first: Why are we still using ECX, not CX? The answer is that even though we are accessing a 16-bit item, its address is still 32 bits.

The corresponding items for 8-bit operations are .byte in place of .long, movb instead of movl, %ah or %al (high and low bytes of AX) in place of EAX, etc.

If you wish to reserve a series of several consecutive bytes and initialize them to a character string, use the .string directive. (The assembler will terminate the string with a null character.) The Intel chips have some special string instructions as well, such as ``stosw.''

6 Linking into an Executable File

In the above example with a source file x.s and object file x.o, we would type

```
ld -o x x.o
```

The linker, **ld**, will link together one or more .o files into an executable file, the name of which is specified by the -o option.

Note that it would not matter whether the x.o file had been produced from Intel-syntax source or AT&T-syntax source. Machine code is machine code, regardless of what generated it.

7 What If You Compile a C Program?

Suppose you type something like

```
gcc y.c
```

Behind the scenes, a lot is happening that is similar to what you see above.

First, **gcc** will temporarily create a .o file, then call **ld** to make an executable file a.out from it, then finally remove the .o file.

Keep in mind that the .o file created from y.c will consist of machine language, just like what we got from running **as** on x.s in our example above. If you want to see the corresponding assembly language, type

```
gcc -S y.c
```

An (AT&T-syntax) assembly language file y.s will be created. You could even then apply **as** to this file, and then run **ld** to create the same executable file **a.out**, though you would also have to link in the proper C library code.

8 How to Execute Those Sample Programs

8.1 ``Normal'' Execution Won't Work

Suppose in our sum-up-4-words example above we name the source file Total.s, and then assemble and link it, with the final executable file named, say, **tot**. We could not simply type

```
tot
```

at the UNIX command line. The program would run correctly, but then it would crash with a segmentation error. Why is this?

The basic problem is that after the last instruction of the program is executed, the processor will attempt to execute the ``instruction'' at the next location of memory. But there is no such instruction, and this fact can be detected by a combination of the hardware and the operating system in various ways. When your program marches right past its last real instruction, a segmentation error will result.

This doesn't happen with your compiled C program, because the compiler inserts a *system call*, i.e. a call to a function in the operating system, which in this case is the exit() call. This results in a graceful transition from your program to the OS, after which the OS prints out your familiar command-line prompt.

We could insert system calls in our sample programs above too, but have not done so because that is a topic to be covered later in the course. Note that that also means no input and output, which is done via system calls too - so, not only does our program crash if we run it in the straightforward manner above, but also we have no way of knowing whether it ran correctly before crashing!

So, in our initial learning environment here, we will execute our programs via a debugger.

8.2 Running Our Programs Using gdb/ddd

8.2.1 Use a Debugging Tool for ALL of Your Programming, in EVERY Class

I've found that many students are **shooting themselves in the foot** by not making use of debugging tools. They learn such a tool in their beginning programming class, but treat it as something that was only to be learned for the final exam, rather than for their own benefit. Subsequently they debug their programs with calls to printf() or cout, which is really a slow, painful way to debug. **You should make use of a debugging tool in all of your programming work - for your benefit, not your professors'.** (See my debugging-tutorial slide show, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`.)

For C/C++ programming on UNIX machines, many debugging tools exist, some of them commercial products, but the most commonly-used one is **gdb**. Actually, many people use **gdb** only indirectly, using **ddd** as their interface to **gdb; ddd** provides a very nice GUI to **gdb**.

8.2.2 Using ddd for Executing Our Assembly Programs

In our case, we will use **gdb** (actually **ddd**, interfacing to **gdb**) to execute our programs. Since it allows us to set breakpoints or single-step through programs, we won't ``go off the end of the earth'' as we would by running our programs directly. Moreover, since the debuggers allow us to inspect registers and memory contents, we can check the ``output'' of our program.

If you have not used **ddd** before, take a few minutes read debugging-tutorial slide show before continuing.

Start by typing, for our summing example above.

```
ddd tot
```

Your source file sum.s should appear in a window. You'll also want windows to display the register contents and memory contents, which you can get as follows:

- To get a register window (which you'll use here), hit Status and then Registers. All registers will be displayed.

  The display includes EFLAGS, the flags register. The Carry Flag is bit 0, i.e. the least-significant bit. The other bits we've studied are the Zero Flag, bit 6, the Sign Flag, bit 7, and the Overflow Flag, bit 11.

- In our case here, we won't need to display memory, e.g. the x array, since x never changes. But in general, to display, say, z, hit Data, then Memory. State how many cells you want displayed, what kind of cells (whole words, individual bytes, etc.), and where to start, such as &z. It will also ask whether you want the information ``printed,'' which means displayed just once, or ``displayed,'' which means a continuing display which reflects the changes to **tot** as the program progresses.

You can now set a breakpoint at the line labeled ``done'', by clicking on that label and then on the stop sign icon.[9] Go ahead and click on Run. After the program stops at that line, see that the contents of EBX is 26, confirming that our program ran correctly.

You can step through your code line by line in the usual debugging-tool manner. However, make sure that you use Nexti and Stepi instead of Next and Step, since we are working at the machine instruction level (the `i' stands for ``instruction.'') Actually, I recommend sticking to Stepi.

Of course, not only can you use the debugger as an ``executor,'' you can also use it as a debugger! Note, by the way, that when you modify your assembly-language source file and reassemble and link, **gdb** or **ddd** will not automatically reload your source and executable files. To reload, click on File in **ddd**, then Open Program and click on the executable file.

8.2.3 Using gdb for Executing Our Assembly Programs

In some cases, you might find it more convenient to use **gdb** directly, rather than via the **ddd** interface. For example, you might be using **telnet**.

Assuming you already know **gdb** (see the link to my Web tutorial below), here are the two new commands you should learn.

- To view register contents, type

  ```
  info registers
  ```

- To view memory, use the **x** (``examine'') command. If for example you have a memory location labeled z and wish to examine the first four words starting at z, type

  ```
  x/4w &z
  ```

  Note that you can use this to examine the stack. To determine the current value in the stack pointer, note first that **ddd** or **gdb** will tell you the values in the registers, including the stack pointer. Say the latter value is 0xbffcec54. Then type

  ```
  (gdb) x/4w 0xbffcec54
  0xbffcec54:     0x080483be      0x00000005      0xbffcec68
  0x00000000
  ```

This gives us the top 4 elements of the stack.

Use **gdb**'s online help facility to get further details.

8.3 An Assembly-Language Specific Debugger: ald

This debugger will serve as your backup in case the -gstabs option in **as** fails on your source code (in which case re-run **as** without the option).

First, start **ald**:

```
ald tot
```

Probably the first thing you will want to do is display your source code:

```
d -s .text
```

(Not shown here.) The display format is address/machine code/assembly language. Note that your source code will be displayed in Intel syntax, whether or not your original file was in that syntax! Also note that the labels don't show up; you just see memory addresses.

To run the program, let's set a breakpoint at the instruction labeled ``done'':

```
break 0x0804808B
r
```

The program stops at the specified point, and the contents of all the registers are automatically displayed. (Not shown here.) Sure enough, we find that EBX has the right value, confirming that the program worked.

If we were to use .data instead of .text above, we would see the contents of that segment (along with bogus ``disassembled instructions''). This is a good way to check memory contents when debugging (or when viewing the ``output'' when using this debugger to execute our program).

Since there are no labels displayed, you will have to work a bit harder in order to set breakpoints or check memory contents. The outputs of the -a option in **as** or the -l option in **nasm** can help you here, as can the output of the UNIX **nm** command, e.g.

```
nm tot
```

Note, though, that addresses may be shifted somewhat when the program is loaded into memory for execution.

Online help is available, by typing ``help'' at the prompt.

9 Useful Web Links

- Unix tutorial: http://heather.cs.ucdavis.edu/~matloff/unix.html
- Linux installation guide: http://heather.cs.ucdavis.edu/~matloff/linux.html
- Linux assembly language Web page: http://linuxassembly.org/
- full **as** manual: http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html (contains full list of directives, register names, etc.; op code names are same as Intel syntax, except for suffixes, e.g. `l' in ``movl'')
- NASM assembler home page: http://nasm.2y.net/
- my tutorials on debugging, featuring my slide show, using **ddd**: http://heather.cs.ucdavis.edu/~matloff/debug.html
- the ALD debugger: http://ellipse.mcs.drexel.edu/ald.html
- the Intel2gas syntax converter: http://www.niksula.cs.hut.fi/~mtiihone/intel2gas/

Footnotes:

[1]One difference, though, is that **as** will merely show addresses as offsets within segment, while **nasm** will show absolute addresses.

[2]By the way, NASM is available for both Unix and MS Windows. For that matter, even **as** can be used under Windows, since it is part of the **gcc** package and that is available for Windows under the name ``cygwin."

[3]The term *long* here is a historical vestige from the old days of 16-bit Intel CPUs. Note that in the Intel syntax the corresponding term is *double*.

[4]Note that x is simply a name for the first word in the array, not the set of 4 words. Knowing this, you should now have some insight into why in C or C++, an array name is synonymous to a pointer to the first element of the array.

[5]We will usually use the present tense in remarks like this, but it should be kept in mind that the action will not actually occur until the program is executed. So, a more precise though rather unwieldy phrasing would be, ``When it is later executed, the **movl** instruction will copy..."

[6] Note that the Intel syntax is quite different. Under that syntax, x would mean the address of x, and the contents of the word x would be denoted as [x].

[7]This presumes that we need it in memory for some other reason. If not, we would not do so.

[8]There are actually a couple of exceptions to this on Intel chips.

[9]For some reason, it will not work if we set a breakpoint at the very first instruction of a program, though any other instruction works.

File translated from TEX by TTH, version 2.88.
On 5 Feb 2002, 15:42.