LINUX VOICE TUTORIAL

ASMSCHOOL: GET STARTED WITH ASSEMBLY LANGUAGE

MIKE SAUNDERS

Part 1: Explore beyond the limits of high-level languages and discover exactly how your CPU really ticks.

WHY DO THIS?

- Learn what compilers do behind the scenes.
- Understand the language of CPUs.
- Fine-tune your code for better performance.

Most people see assembly language as some kind of black magic, part of a dark and scary world where only the top 0.01% of developers ever dare roam. But it's actually a fascinating and surprisingly accessible subject. It's also well worth learning the basics to help you understand how compilers generate code, what CPUs actually do, and get a good all-round picture of how computers work. Assembly language is ultimately a textual representation of the instructions that the CPU executes, with some extra bits 'n bobs to make programming easier.

Nobody in their right mind would write a large desktop application in assembly language today. It'd be monstrously complicated, very hard to debug, and a colossal effort to port to other architectures. But assembly is still used in various places: many drivers in the Linux kernel have chunks written in assembly, partly because it's the best language to use when you're interfacing directly with hardware, and partly for speed reasons. In certain cases, hand-written assembly language can perform better than code generated by a compiler.

Over the next few issues we'll delve into the world of assembly language. We'll explain the basics here, move onto some more advanced logic next month, and finish up with a simple bootable operating system – it won't do a great amount, but it'll be your own code, running on bare hardware, with no other OS required. Sounds good, right? Let's go...

1 YOUR FIRST ASSEMBLY PROGRAM

Many assembly guides start off with huge, bland, tiresome chapters that spend ages talking about binary arithmetic and CPU design theory, before even showing any real code. That's no fun, so we'll get straight into a program. Then we'll go through it step by step so that you can learn how assembly works from a practical example.

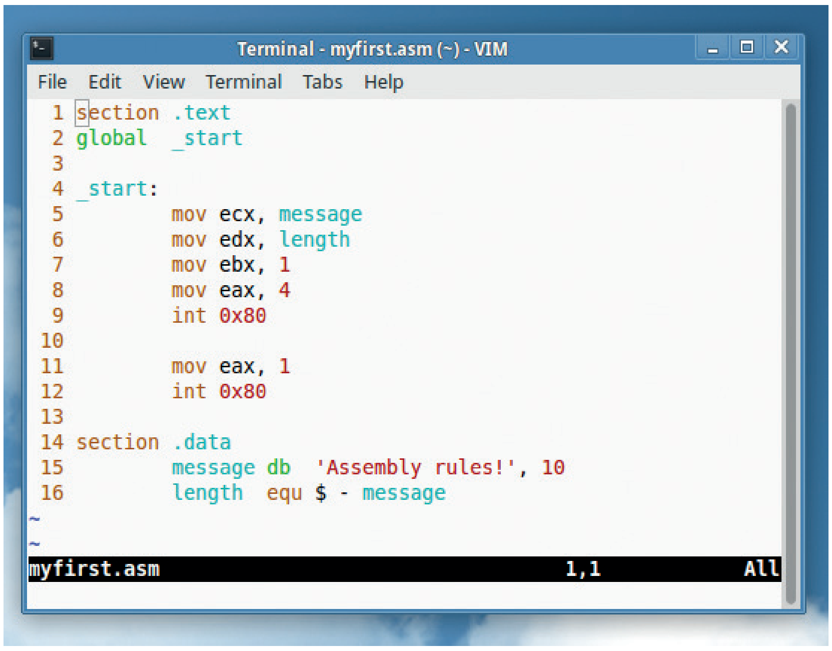
Type this into a plain text editor and save it as **myfirst.asm** in your home directory:

```
section     .text
global      _start

_start:
            mov ecx, message
            mov edx, length
            mov ebx, 1
            mov eax, 4
            int 0x80

            mov eax, 1
            int 0x80

section .data
            message    db  'Assembly rules!', 10
            length     equ $ – message
```

(Note: you can use tabs to indent the code here, or whitespace – it doesn't matter.) This program simply prints the text "Assembly rules!" onto the screen, and then exits.

The tool we're going to use to convert this assembly code into an executable binary file is called, funnily enough, an assembler. There are many assemblers out there, but our favourite is *NASM*; it's available in almost every distro's package repositories, so get it via your graphical package manager, or **yum install nasm**, or **apt-get install nasm**, or whatever's best for your distro.

Now open a terminal window and enter the following commands:


Terminal - myfirst.asm (~) - VIM
File Edit View Terminal Tabs Help

```
section .text
global  _start

_start:
        mov ecx, message
        mov edx, length
        mov ebx, 1
        mov eax, 4
        int 0x80

        mov eax, 1
        int 0x80

section .data
        message db  'Assembly rules!', 10
        length  equ $ - message
~
~
myfirst.asm                                   1,1          All
```

Some text editors, such as **Vim**, include syntax highlighting for assembly (try **set syn=nasm**).

```
nasm -f elf -o myfirst.o myfirst.asm
ld -m elf_i386 -o myfirst myfirst.o
```

The first command here uses *NASM* to generate an object (executable) file called **myfirst.o**, in ELF format (the executable format used on Linux). What exactly is an executable format, you might be asking – why not just use plain binary CPU instructions for the CPU to execute? Well, you could use plain binary back in the 80s, but modern operating systems have more demanding requirements. ELF binaries include information for debugging, they split up code and data into separate sections to stop one from overwriting another, and so forth.

Later in this tutorial series, when we look at writing code to run on bare metal (our mini operating system), we'll explore plain binaries.

A link to the past

So now we have **myfirst.o**, the executable code containing our program. It's not quite complete yet, though; using **ld**, the linker, we link it with some system startup code (ie boilerplate code that's run at the start of every program) to generate an executable called **myfirst**. (The **elf_i386** bit describes the exact binary format – in this case, it means you can use 32-bit assembly code, even if you're running a 64-bit distro.)

If everything has gone smoothly, you can now execute your program as follows:

```
./myfirst
```

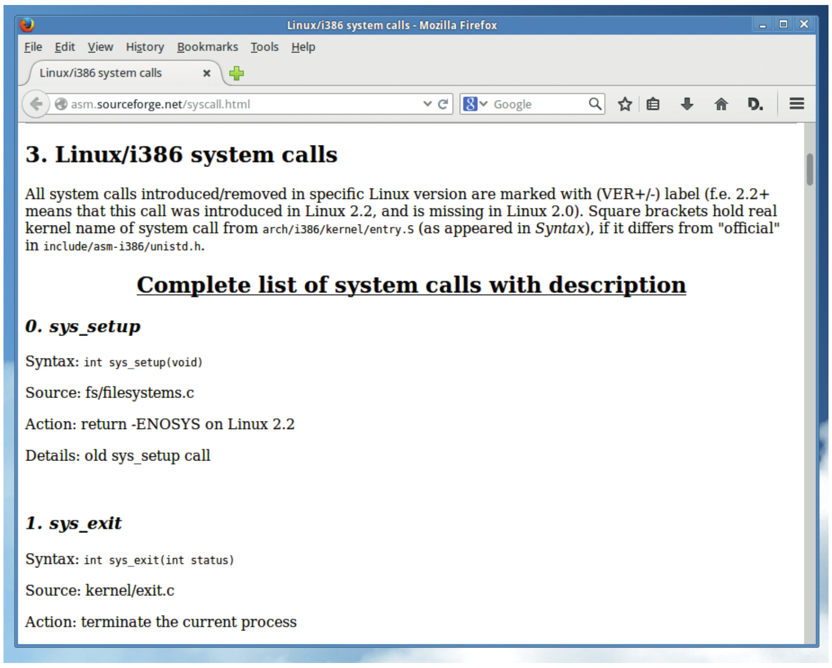
And you should see this output: "Assembly rules!" And there we have it – a complete, standalone Linux program, written entirely in assembly language. Sure, it doesn't do very much, but it's a good way to get started and get an overview of the structure of an assembly program, and see how the source code is converted into binary.

But before we dive into the code itself, it's useful to check out the size of the program. Enter **ls -l myfirst** and you'll see that it's around 670 bytes. Now consider the size of the C equivalent:

```
#include <stdio.h>

int main()
{
	puts("Assembly rules!");
}
```

If you save that as **test.c**, compile it (**gcc -o test test.c**) and look at the resulting **test** binary, you'll see that it's much larger – 8.4k. You can remove some debugging information (**strip -s test**) but it still remains around 6k. This is because *GCC* adds a lot more of the aforementioned boilerplate startup and shutdown code, and also links to a large C library. But it also demonstrates why assembly is the best language to use when space is tight.

Many assembly programmers make good money writing code for extremely restricted embedded devices, for instance, and it's why assembly was the only real choice for writing games back on the old 8-bit consoles and home computers.



Linux/i386 system calls - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Linux/i386 system calls
asm.sourceforge.net/syscall.html
Google

3. Linux/i386 system calls

All system calls introduced/removed in specific Linux version are marked with (VER+/-) label (f.e. 2.2+ means that this call was introduced in Linux 2.2, and is missing in Linux 2.0). Square brackets hold real kernel name of system call from arch/i386/kernel/entry.S (as appeared in *Syntax*), if it differs from "official" in include/asm-i386/unistd.h.

Complete list of system calls with description

0. sys_setup

Syntax: `int sys_setup(void)`

Source: fs/filesystems.c

Action: return -ENOSYS on Linux 2.2

Details: old sys_setup call

1. sys_exit

Syntax: `int sys_exit(int status)`

Source: kernel/exit.c

Action: terminate the current process

Using system calls, you can ask the kernel to perform various tasks relating to files and text input/output.

Disassembling code

Writing new code is fun, but it can be even more interesting to pick apart someone else's work. Using a tool called **objdump** (part of the Binutils package), you can "disassemble" an executable file – essentially turning the binary CPU instructions into their text-mode equivalents. Try it on the **myfirst** example we've been working on in this tutorial, like so:

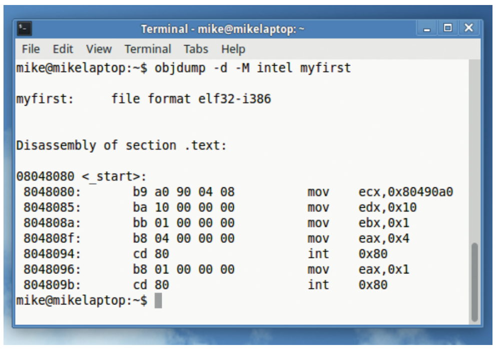
```
objdump -d -M intel myfirst
```

You'll see a list of instructions from the text section of the binary. For instance, the first instruction, where we placed the location of our string into the **ecx** register, looks like this:

```
mov ecx,0x80490a0
```

During assembly, NASM replaced "message" with the actual numerical location of the string in the data section. So disassembled binaries are less useful than the original code, as they're missing things like comments and labels, but they can be useful to see how a program implements a time-critical routine, or performing hacks. Back in the 80s and 90s, many coders used disassembly tools to identify and remove copy protection routines from games, for instance.

You can also disassemble programs written in other languages, but the results can be immensely complicated. Run the above **objdump** command on **/bin/ls**, for instance, and you'll see many thousands of lines of code in the text section, generated by the compiler from the original C source.



Terminal - mike@mikelaptop: ~
File Edit View Terminal Tabs Help

```
mike@mikelaptop:~$ objdump -d -M intel myfirst

myfirst:     file format elf32-i386


Disassembly of section .text:

08048080 <_start>:
 8048080:	b9 a0 90 04 08       	mov    ecx,0x80490a0
 8048085:	ba 10 00 00 00       	mov    edx,0x10
 804808a:	bb 01 00 00 00       	mov    ebx,0x1
 804808f:	b8 04 00 00 00       	mov    eax,0x4
 8048094:	cd 80                	int    0x80
 8048096:	b8 01 00 00 00       	mov    eax,0x1
 804809b:	cd 80                	int    0x80
mike@mikelaptop:~$
```

Here's the disassembly of our sample program, showing hexadecimal codes and the instructions.

2 ANALYSING THE CODE

So, let's now see what each line of our program actually does. We start off with these two:

```
section .text
global  _start
```

These are not CPU instructions, but directives given to the *NASM* assembler; the first one says that the following code belongs in the "text" section of the final executable. Slightly confusingly, the text section doesn't contain plain text (like our "Assembly rules" string), but executable code instead – ie CPU instructions. Next we have **global _start**, which tells the **ld** linker where execution should begin in our file. This is useful if we don't want to start execution right at the beginning of the text section, but somewhere else. The **global** part makes it readable by other tools than just the assembler, so that **ld** can see it.

"Assembly language is really just a bunch of mnemonics for CPU instructions."

So, we've said that execution should begin at the **_start** position. And then we define this location explicitly in our code:

```
_start:
```

Standalone words ending in colons are known as labels, and are locations in the code we can jump to (more on that next issue). So, program execution begins here! And now we come to our first real CPU instruction:

```
mov ecx, message
```

Assembly language is essentially a bunch of mnemonics for CPU instructions (aka machine code). Here, **mov** is one such instruction – it could also be written in the raw binary format that the CPU understands, like 10001011. But working with raw binary would be a nightmare for us puny humans, so we use these slightly more readable variants instead. The assembler simply converts the text instructions to their binary equivalents – although it can do more, as we'll see in later tutorials.

Anyway, to understand this line of code, we also need to understand the concept of registers. CPUs don't do anything especially fancy – they simply move memory around, perform calculations on it, and then do other operations depending on the results. The CPU has no idea what a display is, or a mouse, or a printer. It simply moves data around and performs a few calculations.

Now, the main storage area for data that the CPU uses is your RAM banks. But because RAM is held outside of the CPU, accessing it takes a lot of time. To make things faster and simpler, the CPU includes its own small group of memory cells called registers. CPU instructions can use these registers directly, and in this line of code we're using the register called **ecx**.

This is a 32-bit register (so it can store numbers from zero to 4,294,967,295). You'll see in subsequent lines of code that we also work with **edx**, **ebx** and **eax** – these are all general-purpose registers that we can use for any task, as opposed to special registers that we'll explore next month. And if you're wondering where the names came from: **ecx** started as **c** in the 8-bit days, when was extended to **cx** for 16-bit, and then **ecx** for 32-bit. So the names look a bit odd now, but back on the old CPUs, you had nicely named general-purpose registers like **a**, **b**, **c** and **d**.

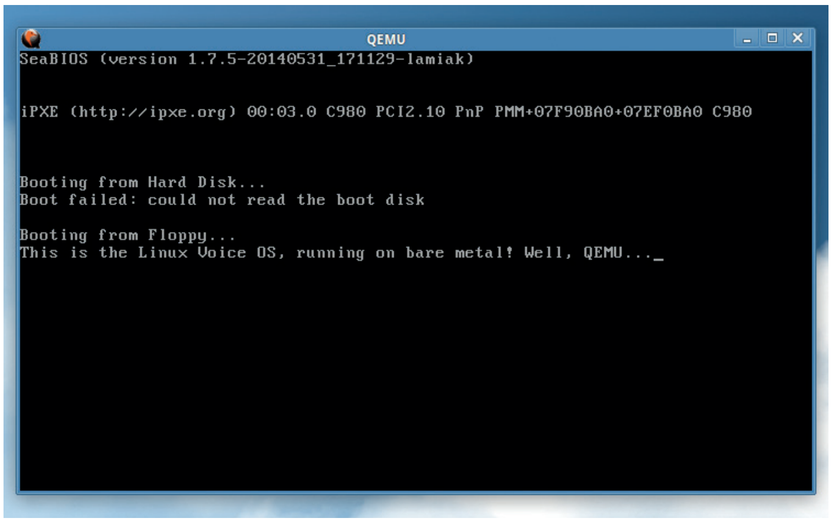
Moving on up

Back to the code: the **mov** instruction moves (actually, copies) a number from one place to another, from right to left. So in this case, we're saying "place message in the **ecx** register". But what is message? It's not another register – it's a location. Down at the bottom of the code, in the data section, you'll see the **message** label followed by **db**, which defines some bytes that are placed at the **message** location in the code. This is really useful, as we don't need to know the exact location of the "Assembly rules" string in the data section – we can just refer to it via the **message** label. (The number 10 after our string is simply a newline character, like adding **\n** to a C string.)

So, we've moved that location into our **ecx** register. But what we're about to do is especially cool. As mentioned, the CPU has no real concept of hardware devices – to print something on the screen, you have to send data to the video card, or move data into video RAM. We have absolutely no idea where this video RAM is, and everyone has a different video card, X server configuration, window manager etc. So directly printing something on the screen is, for us, virtually impossible in a short program.

So what we do is tell the kernel to do it for us. The Linux kernel has a bunch of system calls that low-level programs can use, to get the kernel to do various jobs. One of these calls is to output a string of text. Then the kernel handles it all – and indeed, it offers an even deeper layer of abstraction, in that it can output

And just as a teaser for what's to come: here's bare-metal code, running on a PC emulator! And we'll show you how to boot it on a real box…



```
SeaBIOS (version 1.7.5-20140531_171129-lamiak)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F90BA0+07EF0BA0 C980



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
This is the Linux Voice OS, running on bare metal! Well, QEMU..._
```

Once you pop, you can't stop

One thing we'll be looking at next month is the stack, so we'll get you prepared for it here. The stack is an area of memory where temporary values can be placed, when you need to free up your registers for other purposes. But the most important feature of the stack is the way data is stored in it: you "push" numbers onto the stack, and "pop" them out. It's LIFO (last in, first out), so the most recent item you place onto it is the most recent you'll pull out.

Imagine you have an empty Pringles can, for instance, and you place into it a cream cracker, an Alf Pog and a GameCube disc – in that order. If you now retrieve the items, the first will be the GameCube disc, the second the Alf Pog, and so forth. Here's how it works in assembly language:

```
push 2
push 5
push 10
pop eax
pop ebx
pop ecx
```

After these six instructions, **eax** will contain 10, **ebx** will contain 5, and **ecx** 2. So the stack is a great way to liberate some space temporarily; if you have some important values in **eax** and **ebx**, for instance, but need to do a quick job with them, you can push them onto the stack, do your work, and then pop back the previous values to return to your previous state.

The stack is also used when calling subroutines, in that the return address of the code is pushed onto the stack. This is why you have to be careful when using the stack – if you overwrite it with the wrong data, you might not be able to return to a previous place in the code, and you've taken a one-way trip to crashland!

the string to a plain text terminal, or via a terminal emulator in the X Window System, or even redirect it to an open file.

Before we ask the kernel to print the string, though, we need to provide it with more information than just the string location in the **ecx** register. We also need to tell it how many characters to print, so that it doesn't just keep printing after the end of our string. That's when this line comes into play, in the data section towards the bottom:

```
length equ $ – message
```

Here we have another label, **length**, but instead of using **db** to include some data next to it, we use **equ** to say it's the equivalent of something (a bit like **#define** in C). The dollar sign refers to the current location in the code, so here we're saying: **length** should be equal to the current location in the code, minus the "message" location. In other words, this gives us the length of the "message" string.

Back in the main code, we put this value into the **edx** register:

```
mov edx, length
```

So far so good: two registers are populated with the string location and the number of characters to print. But before we tell the kernel to do its work, we need to provide a bit more information. First, we need to tell the kernel which "file descriptor" to use – in other words, where the output should go to. This topic is beyond the scope of this assembly tutorial, but we need to use **stdout**, which basically means: print to the screen. The number for this is 1, and we put it in the **ebx** register.

Now we're tantalisingly close to using the kernel, but there's one more register to fill. The kernel can actually do lots of different things, such as mounting filesystems, reading data from files, deleting files and so forth. These facilities are provided by the aforementioned system calls, and before we hand control over to the kernel, we need to tell it which call to use. If you look at **http://asm.sourceforge.net/syscall.html**, you'll see some of the many calls available to programs – in our case we want **sys_write** ("write to a file descriptor"), which is number 4. So we place that in the **eax** register:

```
mov eax, 4
```

And that's it! We've set up everything we need to use a kernel system call, so now we hand control over like so:

```
int 0x80
```

Here, **int** stands for "interrupt", and literally interrupts the current program flow by jumping into the kernel. (The 0x80 is hexadecimal here – you don't need to concern yourself with it for now.) The kernel prints the string pointed to in the **ecx** register, then hands control back to our program.

To end our program, we need to call the kernel's **sys_exit** routine, which has the value 1. So we place that number in **eax**, interrupt our program again, and the kernel neatly terminates our program, putting us back at the command prompt. There you have it: a complete (albeit very small) assembly language program, hand written and without the need to use any fat libraries.

We've raced through a lot in this tutorial, and as mentioned, we could've focused entirely on theory instead. But we hope you've found it useful to see a real example in action, and next issue we'll spend more time with some of the concepts we've introduced here. But we'll also take it up a notch, by adding logic and subroutines to our program – assembly versions of IFs and GOTOs.

Meanwhile, as you get familiar with this program, here are some things you can try:

- Print a different, longer string.
- Print two strings, one after another.
- Change the exit code that the program hands back to the shell (this could take a bit of Googling!).

If you get stuck or need any help, pop by our forums on **http://forums.linuxvoice.com** – this author will be at hand to point you in the right direction. Happy hacking! LV

Mike Saunders has written a whole OS in assembly (http://mikeos.sf.net) and is contemplating a Pi version.