Programming from the Ground Up

Jonathan Bartlett

Edited by
Dominick Bruno, Jr.

Programming from the Ground Up

by Jonathan Bartlett

Edited by Dominick Bruno, Jr.

This book can be purchased in paperback at http://www.bartlettpublishing.com/

This book is not a reference book, it is an introductory book. It is therefore not suitable by itself to learn how to professionally program in x86 assembly language, as some details have been left out to make the learning process smoother. The point of the book is to help the student understand how assembly language and computer programming works, not to be a definitive reference to the subject.

To receive a copy of this book in electronic form, please visit the website http://savannah.nongnu.org/projects/pgubook/ This site contains the instructions for downloading a transparent copy (as defined by the GNU FDL) of this book.

Table of Contents

Chapter 1. Introduction

Welcome to Programming

I love programming. I enjoy the challenge to not only make a working program, but to do so with style. Programming is like poetry. It conveys a message, not only to the computer, but to those who modify and use your program. With a program, you build your own world with your own rules. You create your world according to your conception of both the problem and the solution. Masterful programmers create worlds with programs that are clear and succinct, much like a poem or essay.

One of the greatest programmers, Donald Knuth, describes programming not as telling a computer how to do something, but telling a person how they would instruct a computer to do something. The point is that programs are meant to be read by people, not just computers. Your programs will be modified and updated by others long after you move on to other projects. Thus, programming is not as much about communicating to a computer as it is communicating to those who come after you. A programmer is a problem-solver, a poet, and an instructor all at once. Your goal is to solve the problem at hand, doing so with balance and taste, and teach your solution to future programmers. I hope that this book can teach at least some of the poetry and magic that makes computing exciting.

Most introductory books on programming frustrate me to no end. At the end of them you can still ask "how does the computer really work?" and not have a good answer. They tend to pass over topics that are difficult even though they are important. I will take you through the difficult issues because that is the only way to move on to masterful programming. My goal is to take you from knowing nothing about programming to understanding how to think, write, and learn like a programmer. You won't know everything, but you will have a background for how everything fits together. At the end of this book, you should be able to do the following:

- Understand how a program works and interacts with other programs
- Read other people's programs and learn how they work
- Learn new programming languages quickly
- Learn advanced concepts in computer science quickly

I will not teach you everything. Computer science is a massive field, especially when you combine the theory with the practice of computer programming. However, I will attempt to get you started on the foundations so you can easily go wherever you want afterwards.

There is somewhat of a chicken and egg problem in teaching programming, especially assembly language. There is a lot to learn - it's almost too much to learn almost at once, but each piece

depends on all the others. Therefore, you must be patient with yourself and the computer while learning to program. If you don't understand something the first time, reread it. If you still don't understand it, it is sometimes best to take it by faith and come back to it later. Often after more exposure to programming the ideas will make more sense. Don't get discouraged. It's a long climb, but very worthwhile.

At the end of each chapter are three sets of review exercises. The first set is more or less regurgitation - they check to see if can you give back what you learned in the chapter. The second set contains application questions - they check to see if you can apply what you learned to solve problems. The final set is to see if you are capable of broadening your horizons. Some of these questions may not be answerable until later in the book, but they give you some things to think about. Other questions require some research into outside sources to discover the answer. Still others require you to simply analyze your options and explain a best solution. Many of the questions don't have right or wrong answers, but that doesn't mean they are unimportant. Learning the issues involved in programming, learning how to research answers, and learning how to look ahead are all a major part of a programmer's work.

If you have problems that you just can't get past, there is a mailing list for this book where readers can discuss and get help with what they are reading. The address is `pgubook-readers@nongnu.org`. This mailing list is open for any type of question or discussion along the lines of this book.

Your Tools

This book teaches assembly language for x86 processors and the GNU/Linux operating system. Therefore we will be giving all of the examples using the GNU/Linux standard GCC tool set. If you are not familiar with GNU/Linux and the GCC tool set, they will be described shortly. If you are new to Linux, you should check out the guide available at http://rute.sourceforge.net/[1] What I intend to show you is more about programming in general than using a specific tool set on a specific platform, but standardizing on one makes the task much easier.

Those new to Linux should also try to get involved in their local GNU/Linux User's Group. User's Group members are usually very helpful for new people, and will help you from everything from installing Linux to learning to use it most efficiently. A listing of GNU/Linux User's Groups is available at http://www.linux.org/groups/

All of these programs have been tested using Red Hat Linux 8.0, and should work with any other GNU/Linux distribution, too.[2] They will not work with non-Linux operating systems such as

1. This is quite a large document. You certainly don't need to know everything to get started with this book. You simply need to know how to navigate from the command line and how to use an editor like `pico`, `emacs`, or `vi` (or others).
2. By "GNU/Linux distribution", I mean an x86 GNU/Linux distribution. GNU/Linux distributions for the Power Macintosh, the Alpha processor, or other processors will not work with this book.

BSD or other systems. However, all of the *skills* learned in this book should be easily transferable to any other system.

If you do not have access to a GNU/Linux machine, you can look for a hosting provider who offers a Linux *shell account*, which is a command-line only interface to a Linux machine. There are many low-cost shell account providers, but you have to make sure that they match the requirements above (i.e. - Linux on x86). Someone at your local GNU/Linux User's Group may be able to give you one as well. Shell accounts only require that you already have an Internet connection and a telnet program. If you use Windows, you already have a telnet client - just click on `start`, then `run`, then type in `telnet`. However, it is usually better to download PuTTY from http://www.chiart.greenend.co.uk/~sgtatham/putty/ because Windows' telnet has some weird problems. There are a lot of options for the Macintosh, too. NiftyTelnet is my favorite.

If you don't have GNU/Linux and can't find a shell account service, then you can download Knoppix from http://www.knoppix.org/ Knoppix is a GNU/Linux distribution that boots from CD so that you don't have to actually install it. Once you are done using it, you just reboot and remove the CD and you are back to your regular operating system.

So what is GNU/Linux? GNU/Linux is an operating system modeled after UNIX. The GNU part comes from the GNU Project (http://www.gnu.org/)[3], which includes most of the programs you will run, including the GCC tool set that we will use to program with. The GCC tool set contains all of the programs necessary to create programs in various computer languages.

Linux is the name of the *kernel*. The kernel is the core part of an operating system that keeps track of everything. The kernel is both an fence and a gate. As a gate, it allows programs to access hardware in a uniform way. Without the kernel, you would have to write programs to deal with every device model ever made. The kernel handles all device-specific interactions so you don't have to. It also handles file access and interaction between processes. For example, when you type, your typing goes through several programs before it hits your editor. First, the kernel is what handles your hardware, so it is the first to receive notice about the keypress. The keyboard sends in *scancodes* to the kernel, which then converts them to the actual letters, numbers, and symbols they represent. If you are using a windowing system (like Microsoft Windows or the X Window System), then the windowing system reads the keypress from the kernel, and delivers it to whatever program is currently in focus on the user's display.

Example 1-1. How the computer processes keyboard sigals

```
Keyboard -> Kernel -> Windowing system -> Application program
```

The kernel also controls the flow of information between programs. The kernel is a program's gate to the world around it. Every time that data moves between processes, the kernel controls the

3. The GNU Project is a project by the Free Software Foundation to produce a complete, free operating system.

messaging. In our keyboard example above, the kernel would have to be involved for the windowing system to communicate the keypress to the application program.

As a fence, the kernel prevents programs from accidentally overwriting each other's data and from accessing files and devices that they don't have permission to. It limits the amount of damage a poorly-written program can do to other running programs.

In our case, the kernel is Linux. Now, the kernel all by itself won't do anything. You can't even boot up a computer with just a kernel. Think of the kernel as the water pipes for a house. Without the pipes, the faucets won't work, but the pipes are pretty useless if there are no faucets. Together, the user applications (from the GNU project and other places) and the kernel (Linux) make up the entire operating system, GNU/Linux.

For the most part, this book will be using the computer's low-level assembly language. There are essentially three kinds of languages:

Machine Language

> This is what the computer actually sees and deals with. Every command the computer sees is given as a number or sequence of numbers.

Assembly Language

> This is the same as machine language, except the command numbers have been replaced by letter sequences which are easier to memorize. Other small things are done to make it easier as well.

High-Level Language

> High-level languages are there to make programming easier. Assembly language requires you to work with the machine itself. High-level languages allow you to describe the program in a more natural language. A single command in a high-level language usually is equivalent to several commands in an assembly language.

In this book we will learn assembly language, although we will cover a bit of high-level languages.

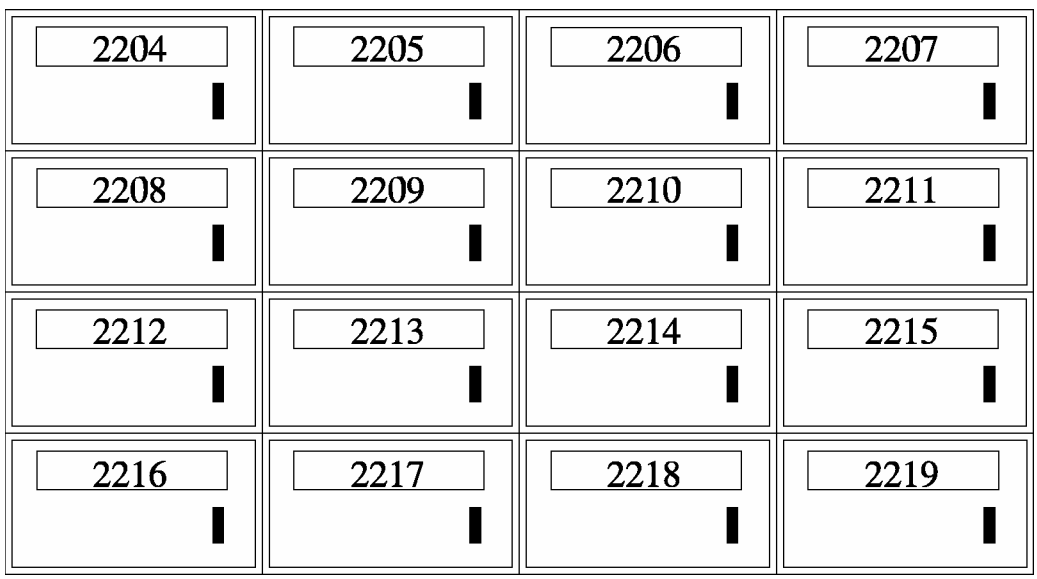
Chapter 2. Computer Architecture

Before learning how to program, you need to first understand how a computer interprets programs. You don’t need a degree in electrical engineering, but you need to understand some basics.

Modern computer architecture is based off of an architecture called the Von Neumann architecture, named after its creator. The Von Neumann architecture divides the computer up into two main parts - the CPU (for Central Processing Unit) and the memory. This architecture is used in all modern computers, including personal computers, supercomputers, mainframes, and even cell phones.

Structure of Computer Memory

To understand how the computer views memory, imagine your local post office. They usually have a room filled with PO Boxes. These boxes are similar to computer memory in that each are numbered sequences of fixed-size storage locations. For example, if you have 256 megabytes of computer memory, that means that your computer contains roughly 256 million fixed-size storage locations. Or, to use our analogy, 256 million PO Boxes. Each location has a number, and each location has the same, fixed-length size. The difference between a PO Box and computer memory is that you can store all different kinds of things in a PO Box, but you can only store a single number in a computer memory storage location.



Memory locations are like PO Boxes

You may wonder why a computer is organized this way. It is because it is simple to implement. If

the computer were composed of a lot of differently-sized locations, or if you could store different kinds of data in them, it would be difficult and expensive to implement.

The computer's memory is used for a number of different things. All of the results of any calculations are stored in memory. In fact, everything that is "stored" is stored in memory. Think of your computer at home, and imagine what all is stored in your computer's memory.

- The location of your cursor on the screen
- The size of each window on the screen
- The shape of each letter of each font being used
- The layout of all of the controls on each window
- The graphics for all of the toolbar icons
- The text for each error message and dialog box
- The list goes on and on...

In addition to all of this, the Von Neumann architecture specifies that not only computer data should live in memory, but the programs that control the computer's operation should live there, too. In fact, in a computer, there is no difference between a program and a program's data except how it is used by the computer. They are both stored and accessed the same way.

The CPU

So how does the computer function? Obviously, simply storing data doesn't do much help - you need to be able to access, manipulate, and move it. That's where the CPU comes in.

The CPU reads in instructions from memory one at a time and executes them. This is known as the *fetch-execute cycle*. The CPU contains the following elements to accomplish this:

- Program Counter
- Instruction Decoder
- Data bus
- General-purpose registers
- Arithmetic and logic unit

The *program counter* is used to tell the computer where to fetch the next instruction from. We mentioned earlier that there is no difference between the way data and programs are stored, they are just interpreted differently by the CPU. The program counter holds the memory address of the next instruction to be executed. The CPU begins by looking at the program counter, and fetching

whatever number is stored in memory at the location specified. It is then passed on to the *instruction decoder* which figures out what the instruction means. This includes what process needs to take place (addition, subtraction, multiplication, data movement, etc.) and what memory locations are going to be involved in this process. Computer instructions usually consist of both the actual instruction and the list of memory locations that are used to carry it out.

Now the computer uses the *data bus* to fetch the memory locations to be used in the calculation. The data bus is the connection between the CPU and memory. It is the actual wire that connects them. If you look at the motherboard of the computer, the wires that go out from the memory are your data bus.

In addition to the memory on the outside of the processor, the processor itself has some special, high-speed memory locations called registers. There are two kinds of registers - *general registers* and *special-purpose registers*. General-purpose registers are where the main action happens. Addition, subtraction, multiplication, comparisions, and other operations generally use general-purpose registers for processing. However, computers have very few general-purpose registers. Most information is stored in main memory, brought in to the registers for processing, and then put back into memory when the processing is completed. *special-purpose registers* are registers which have very specific purposes. We will discuss these as we come to them.

Now that the CPU has retrieved all of the data it needs, it passes on the data and the decoded instruction to the *arithmetic and logic unit* for further processing. Here the instruction is actually executed. After the results of the computation have been calculated, the results are then placed on the data bus and sent to the appropriate location in memory or in a register, as specified by the instruction.

This is a very simplified explanation. Processors have advanced quite a bit in recent years, and are now much more complex. Although the basic operation is still the same, it is complicated by the use of cache hierarchies, superscalar processors, pipelining, branch prediction, out-of-order execution, microcode translation, coprocessors, and other optimizations. Don’t worry if you don’t know what those words mean, you can just use them as Internet search terms if you want to learn more about the CPU.

Some Terms

Computer memory is a numbered sequence of fixed-size storage locations. The number attached to each storage location is called it’s *address*. The size of a single storage location is called a *byte*. On x86 processors, a byte is a number between 0 and 255.

You may be wondering how computers can display and use text, graphics, and even large numbers when all they can do is store numbers between 0 and 255. First of all, specialized hardware like graphics cards have special interpretations of each number. When displaying to the

screen, the computer uses ASCII code tables to translate the numbers you are sending it into letters to display on the screen, with each number translating to exactly one letter or numeral.[1] For example, the capital letter A is represented by the number 65. The numeral 1 is represented by the number 49. So, to print out "HELLO", you would actually give the computer the sequence of numbers 72, 69, 76, 76, 79. To print out the number 100, you would give the computer the sequence of numbers 49, 48, 48. A list of ASCII characters and their numeric codes is found in Appendix D.

In addition to using numbers to represent ASCII characters, you as the programmer get to make the numbers mean anything you want them to, as well. For example, if I am running a store, I would use a number to represent each item I was selling. Each number would be linked to a series of other numbers which would be the ASCII codes for what I wanted to display when the items were scanned in. I would have more numbers for the price, how many I have in inventory, and so on.

So what about if we need numbers larger than 255? We can simply use a combination of bytes to represent larger numbers. Two bytes can be used to represent any number between 0 and 65536. Four bytes can be used to represent any number between 0 and 4294967295. Now, it is quite difficult to write programs to stick bytes together to increase the size of your numbers, and requires a bit of math. Luckily, the computer will do it for us for numbers up to 4 bytes long. In fact, four-byte numbers are what we will work with by default.

We mentioned earlier that in addition to the regular memory that the computer has, it also has special-purpose storage locations called *registers*. Registers are what the computer uses for computation. Think of a register as a place on your desk - it holds things you are currently working on. You may have lots of information tucked away in folders and drawers, but the stuff you are working on right now is on the desk. Registers keep the contents of numbers that you are currently manipulating.

On the computers we are using, registers are each four bytes long. The size of a typical register is called a computer's *word* size. x86 processors have four-byte words. This means that it is most natural on these computers to do computations four bytes at a time. This gives us roughly 4 billion values.

Addresses are also four bytes (1 word) long, and therefore also fit into a register. x86 processors can access up to 4294967296 bytes if enough memory is installed. Notice that this means that we can store addresses the same way we store any other number. In fact, the computer can't tell the difference between a value that is an address, a value that is a number, a value that is an ASCII code, or a value that you have decided to use for another purpose. A number becomes an ASCII code when you attempt to display it. A number becomes an address when you try to look up the

1. With the advent of international character sets and Unicode, this is not entirely true anymore. However, for the purposes of keeping this simple for beginners, we will use the assumption that one number translates directly to one character. For more information, see Appendix D.

byte it points to. Take a moment to think about this, because it is crucial to understanding how computer programs work.

Addresses which are stored in memory are also called *pointers*, because instead of having a regular value in them, they point you to a different location in memory.

As we've mentioned, computer instructions are also stored in memory. In fact, they are stored exactly the same way that other data is stored. The only way the computer knows that a memory location is an instruction is that a special-purpose register called the instruction pointer points to them at one point or another. If the instruction pointer points to a memory word, it is loaded as an instruction. Other than that, the computer has no way of knowing the difference between programs and other types of data.[2]

Interpreting Memory

Computers are very exact. Because they are exact, programmers have to be equally exact. A computer has no idea what your program is supposed to do. Therefore, it will only do exactly what you tell it to do. If you accidentally print out a regular number instead of the ASCII codes that make up the number's digits, the computer will let you - and you will wind up with jibberish on your screen (it will try to look up what your number represents in ASCII and print that). If you tell the computer to start executing instructions at a location containing data instead of program instructions, who knows how the computer will interpret that - but it will certainly try. The computer will execute your instructions in the exact order you specify, even if it doesn't make sense.

The point is, the computer will do exactly what you tell it, no matter how little sense it makes. Therefore, as a programmer, you need to know exactly how you have your data arranged in memory. Remember, computers can only store numbers, so letters, pictures, music, web pages, documents, and anything else are just long sequences of numbers in the computer, which particular programs know how to interpret.

For example, say that you wanted to store customer information in memory. One way to do so would be to set a maximum size for the customer's name and address - say 50 ASCII characters for each, which would be 50 bytes for each. Then, after that, have a number for the customer's age and their customer id. In this case, you would have a block of memory that would look like this:

```
Start of Record:
     Customer's name (50 bytes) - start of record
     Customer's address (50 bytes) - start of record + 50 bytes
     Customer's age (1 word - 4 bytes) - start of record + 100 bytes
```

2. Note that here we are talking about general computer theory. Some processors and operating systems actually mark the regions of memory that can be executed with a special marker that indicates this.

```
     Customer's id number (1 word - 4 bytes) - start of record + 104 bytes
```

This way, given the address of a customer record, you know where the rest of the data lies. However, it does limit the customer's name and address to only 50 ASCII characters each.

What if we didn't want to specify a limit? Another way to do this would be to have in our record pointers to this information. For example, instead of the customer's name, we would have a pointer to their name. In this case, the memory would look like this:

```
Start of Record:
     Customer's name pointer (1 word) - start of record
     Customer's address pointer (1 word) - start of record + 4
     Customer's age (1 word) - start of record + 8
     Customer's id number (1 word) - start of record + 12
```

The actual name and address would be stored elsewhere in memory. This way, it is easy to tell where each part of the data is from the start of the record, without explicitly limitting the size of the name and address. If the length of the fields within our records could change, we would have no idea where the next field started. Because records would be different sizes, it would also be hard to find where the next record began. Therefore, almost all records are of fixed lengths. Variable-length data is usually store separately from the rest of the record.

Data Accessing Methods

Processors have a number of different ways of accessing data, known as addressing modes. The simplest mode is *immediate mode*, in which the data to access is embedded in the instruction itself. For example, if we want to initialize a register to 0, instead of giving the computer an address to read the 0 from, we would specify immediate mode, and give it the number 0.

In the *register addressing mode*, the instruction contains a register to access, rather than a memory location. The rest of the modes will deal with addresses.

In the *direct addressing mode*, the instruction contains the memory address to access. For example, I could say, please load this register with the data at address 2002. The computer would go directly to byte number 2002 and copy the contents into our register.

In the *indexed addressing mode*, the instruction contains a memory address to access, and also specifies an *index register* to offset that address. For example, we could specify address 2002 and an index register. If the index register contains the number 4, the actual address the data is loaded from would be 2006. This way, if you have a set of numbers starting at location 2002, you can cycle between each of them using an index register. On x86 processors, you can also specify a *multiplier* for the index. This allows you to access memory a byte at a time or a word at a time (4 bytes). If you are accessing an entire word, your index will need to be multiplied by 4 to get the

exact location of the fourth element from your address. For example, if you wanted to access the fourth byte from location 2002, you would load your index register with 3 (remember, we start counting at 0) and set the multiplier to 1 since you are going a byte at a time. This would get you location 2005. However, if you wanted to access the fourth word from location 2002, you would load your index register with 3 and set the multiplier to 4. This would load from location 2014 - the fourth word. Take the time to calculate these yourself to make sure you understand how it works.

In the *indirect addressing mode*, the instruction contains a register that contains a pointer to where the data should be accessed. For example, if we used indirect addressing mode and specified the `%eax` register, and the `%eax` register contained the value 4, whatever value was at memory location 4 would be used. In direct addressing, we would just load the value 4, but in indirect addressing, we use 4 as the address to use to find the data we want.

Finally, there is the *base-pointer addressing mode*. This is similar to indirect addressing, but you also include a number called the *offset* to add to the register's value before using it for lookup. We will use this mode quite a bit in this book.

In the Section called *Interpreting Memory* we discussed having a structure in memory holding customer information. Let's say we wanted to access the customer's age, which was the eighth byte of the data, and we had the address of the start of the structure in a register. We could use base-pointer addressing and specify the register as the base pointer, and 8 as our offset. This is a lot like indexed addressing, with the difference that the offset is constant and the pointer is held in a register, and in indexed addressing the offset is in a register and the pointer is constant.

There are other forms of addressing, but these are the most important ones.

Review

Know the Concepts

- Describe the fetch-execute cycle.
- What is a register? How would computation be more difficult without registers?
- How do you represent numbers larger than 255?
- How big are the registers on the machines we will be using?
- How does a computer know how to interpret a given byte or set of bytes of memory?
- What are the addressing modes and what are they used for?
- What does the instruction pointer do?