Preface

This book introduces the concepts of how computer hardware works from a programmer‘s point of view. A programmer‘s job is to design a sequence of instructions that will cause the hardware to perform operations that solve a problem. This book looks at these instructions by exploring how C/C++ language constructs are implemented at the instruction set architecture level.

The specific architecture presented in this book is the x86-64 that has evolved over the years from the Intel 8086 processor. The GNU programming environment is used, and the operating system kernel is Linux.

The basic guidelines I followed in creating this book are:

- One should avoid writing in assembly language except when absolutely necessary.
- Learning is easier if it builds upon concepts you already know.
- “Real world” hardware and software make a more interesting platform for learning theoretical concepts.
- The tools used for teaching should be inexpensive and readily available.

It may seem strange that I would recommend against assembly language programming in a book largely devoted to the subject. Well, C was introduced in 1978 specifically for low-level programming. C code is much easier to write and to maintain than assembly language. C compilers have evolved to a point where they produce better machine code than all but the best assembly language programmers can. In addition, the hardware technology has increased such that there is seldom any significant advantage in writing the most efficient machine code. In short, it is hardly ever worth the effort to write in assembly language.

You might well ask why you should study assembly language, given that I think you should avoid writing in it. I believe very strongly that the best programmers have a good understanding of how computer hardware works. I think this principle holds in most fields: the best drivers understand how automobiles work; the best musicians understand how their instrument works; etc.

So this is *not* a book on how to write programs in assembly language. Most of the programs you will be asked to write will be in assembly language, but they are very simple programs intended to illustrate the concepts. I believe that this book will help you to become a better programmer in any programming language, even if you never write another line of assembly language.

Two issues arise immediately when studying assembly language:

- I/O interaction with a user through even the keyboard and screen is a very complex problem, well beyond the programming expertise of a beginner.
- There is an almost endless variety of instructions that can be used.

There are several ways to deal with these problems in a textbook. Some books use a simple operating system for I/O, e.g., MS-DOS. Others provide libraries of I/O functions that are specific for the examples in the book. Several textbooks deal with the instruction set issue by presenting a simplified “idealized” architecture with a small number of instructions that is intended to illustrate the concepts.

In keeping with the “real world” criterion of this book, it deals with these two issues by:

1. showing you how to call the I/O functions already available in the C Standard Library, and
2. presenting only a small subset of the available instructions.

This has the additional advantage of not requiring additional software to be installed. In general, all the programming discussed in the book and be done on any of the common Linux distributions that has been set up for software development with few or no changes.

Readers who wish to write assembly language programs that do not use the C runtime environment should read Sections 8.6 (page 587) and 15.6 (page 875).

If you do decide to write more complex programs in assembly language there are several other excellent books on that topic; see the Bibliography on page 992. And, of course, you would want the manufacturer's programming manuals; see for example [2] – [6] and [14] – [18]. The goal here is to provide you with an introductory "look under the hood" of a high-level language at the hardware that lies below.

This book also provides an introduction to computer hardware architecture. The view is from a programmer's eye. Other excellent books provide implementation details. You need to understand many of the implementation details, e.g., pipelining, caches, in order to write highly optimized programs. This book provides the introduction that prepares you for learning about more advanced architectural concepts.

This is not the place to argue about operating systems. I could rationalize my choice of GNU/Linux, but I could also rationalize using others. Therefore, I will simply state that I believe that GNU/Linux provides an excellent environment for studying programming in an academic setting. One of the more important features of the GNU programming environment with respect to the goals of this book is the close integration of C/C++ and assembly language. In addition, I like GNU/Linux.

I wish to comment on my use of "GNU/Linux" instead of the simpler "Linux." Much has been written about these names. A good source of the various arguments can be found at `www.wikipedia.org`. The two main points are that (a) Linux is only the kernel, and (b) all general-purpose distributions rely on many GNU components for the remaining systems software. Although "Linux" has become essentially a synonym for "GNU/Linux," this book could not exist without the GNU components, e.g., the assembler (`as`), the link editor (`ld`), the `make` program, etc. Therefore, I wish to acknowledge the importance of the GNU project by using the full "GNU/Linux" name.

In some ways, the x86-64 instruction set architecture is not the best choice for studying computer architecture. It maintains backwards compatibility and is thus somewhat more complicated at the instruction set level. However, it is by far the most widely deployed architecture on the desktop and one of the least expensive way to set up a system where these concepts can be studied.

Assembly language is my favorite subject in computer science, but I have taught the subject to enough students to know that, realistically, it probably will not be the same for you. However, please keep your eye on the long term. I am confident that material presented in this book will help you to become a better programmer, and if you do enjoy assembly language, you will have a good introduction to a more advanced study of it.

Assumed Background

You should have taken an introductory class in programming, preferably in C, C++, or Java. The high-level language used in this book is C, however all the C programming is simple. I am confident that the C programming examples in Chapters 2 and 3 will provide sufficient C programming concepts to make the rest of the book very usable, regardless of the language you learned in your introductory class.

I believe that more experienced programmers who wish to write for the x86-64 architecture can also benefit from reading this book. In principle, these programmers can learn everything they need to know from reading the appropriate manuals. However, I have found that it is usually helpful to have an overview of a new architecture before tackling the manuals. This book should provide that overview. In this sense, I believe that this book can provide a good "introduction" to using the manuals.

Additional Resources

I maintain additional resources related to this book, including an errata, on my website, `bob.cs.sonoma.edu`. I welcome your feedback (`plantz@sonoma.edu`), especially any errors or confusing writing that you see in the book. I use such feedback, mostly from students, to constantly improve the book.

Learning from this Book

This book is intended for a one-semester, four unit course. Our course format at Sonoma State University consists of three hours of lecture and a two – three hour supervised lab session per week. Many of the exercises in each chapter provide good in-lab exercises for supervised labs.

Solutions to almost all the chapter exercises are provided in Appendix E. Students should attempt to solve an exercise *before* looking at the answer for hints. But I think it helps the learning process if a student can see a solution while attempting his or her own solution.

If you have an electronic copy of this book, do *not* copy and paste code. Think about it — typing in the code forces you to read every single character. Yes, it is very tedious, but you will learn much more this way. I'm assuming here that your goal is to learn the material, not simply to get the example programs to work. They are rather silly programs, so just getting them to work is not of much use.

Development Environment

Most developers use an Integrated Development Environment (IDE), which hides the process of building a program from source code. In this book we use the component programs individually so that you can see what is taking place.

The examples in this book were compiled or assembled on a computer running Ubuntu 12.04. The development programs used were:

- `gcc` version 4.7.0
- `as` version 2.22

In most cases compilation was done with no optimization (`-O0`) because the goal is to study concepts, not create the most efficient code.

The examples should work in any x86_64 GNU development environment with `gcc` and `as` (`binutils`) installed. However, the machine code generated by the compiler may differ depending on its specific configuration and version. You will begin looking at compiler-generated assembly language in Chapter 7. What you see in your environment may differ from the examples in this book, but the differences should be consistent as you continue through the rest of the book.

You should also keep in mind that the programs used for development may have bugs. Yes, nobody is perfect. For example, when I upgraded my Ubuntu system from 9.04 to 9.10, the GNU assembler was upgraded from 2.19 to 2.20. The newer version had a bug that caused the line numbering in a particular listing file to start from 0 instead of 1. (It affected the C source code in Listing 7.6 on page 513; the numbers have been corrected in this listing.) Fortunately, this bug did not affect the quality of the final program, but it could cause some confusion to the programmer.

Organization of the Book

Data storage formats are covered in Chapters 2 and 3. Chapter 2 introduces the binary and hexadecimal number systems and presents the ASCII code for storing character data. Decimal integers, both signed and unsigned, are discussed in Chapter 3 along with the code used to store them. We use C programs to explore the concepts in Chapter 3. The C examples also provide an introduction to programming in C for those who have not used it yet. This introduction to C will be sufficient for the rest of the book.

Chapters 4 and 5 get down to the actual hardware level. Chapter 4 introduces the mathematics and electronic circuits used to build computers. There is a section on basic electronic circuit elements for those who are new to electronics. Then Chapter 5 moves on to some of the more common logic circuits used in computers. It ends with a discussion of memory implementations. If the book is being used for a software-only course, the instructor could consider skipping over these two chapters

Chapter 6 introduces the central processing unit (CPU) and its relationship to memory and I/O. There is a description of how to use the `gdb` debugger to view the registers in the CPU. The basic set of registers used by programmers in the x86-64 architecture is given in this chapter.

Assembly language programming is introduced in Chapter 7. The topic is introduced by showing how to create a file containing the assembly language generated by the `gcc` compiler from C code. The basic assembly language template for a function is introduced, both for 64-bit and 32-bit mode. There is an overall sketch of how assemblers and linkers work.

In Chapter 8 we see how automatic variables are allocated on the stack, how values are assigned to them, and how functions are called. Argument passing, both in registers and on the stack, is discussed. The chapter shows how to call the `write`, `read`, `printf`, and `scanf` C Standard Library functions for user I/O. There is also a section on writing standalone programs that do not use the C environment and use the `syscall` instruction for direct operating system I/O.

Chapter 9 gives an introduction to machine code. There is a discussion of the REX codes used in 64-bit mode. Two instructions, `mov` and `add`, are used as examples.

Program control flow, specifically repetition and binary decision, are covered in in Chapter 10. Conditional jumps are discussed in this chapter.

Chapter 11 discusses how to write your own functions and use the arguments passed to it. Both the 64-bit and 32-bit function interface techniques are described.

Bit-level logical and shift operations are covered in Chapter 12. The multiplication and division instructions are also discussed.

Arrays and `structs` are discussed in Chapter 13. This chapter includes a discussion of how simple C++ objects are implemented at both the C and the assembly language level.

Until this point in the book we have been using integers. In Chapter 14 we introduce formats for storing fractional values, including some IEEE 754 formats. In 64-bit mode the `gcc` compiler uses SSE2 instructions for floating point, but x87 instructions are used in 32-bit mode. The chapter gives an introduction to both instruction sets.

Exceptions and interrupts are discussed in Chapter 15. Chapter 16 is an introduction to hardware level I/O. Since most students will never do I/O at this level, this is another chapter that could be skipped.

A summary of the instructions used in this book is provided in Appendix A.5. At this point, there is only a list of the instructions. Eventually, there will be a description of each of them.

Appendix B is a highly simplified discussion of the fundamental concepts of the `make` facility.

Appendix C provides a very brief tutorial on using `gdb` for assembly language programs.

Appendix D gives a very brief introduction to the `gcc` syntax for embedding assembly language in a C function.

Almost all the solutions to the chapter exercises are provided in Appendix E. These can be useful for students who wish to use the exercises for self study; if you find yourself getting stuck on a problem, peek at the solution for some hints. Instructors are encouraged to discuss these solutions with their students. There is much to be learned from looking at another person's solution and thinking about how you might do it better.

The Bibliography lists a small fraction of the many books I have consulted when learning this material. I urge you to look at this list of books. I believe that you will want at least some of them in your reference library.

Suggested Usage

- Our course at Sonoma State University covers each chapter approximately in the book's order. The programming exercises in Chapters 2 and 3 get the students used to using the lab right from the beginning of the course. Hardware simulators are used in the lab for Chapters 4 and 5.
- A pure assembly language course could easily omit Chapters 4 and 5.
- In a curriculum where binary numbers are covered in another course Chapters 2 and 3 could be skimmed. I recommend covering the C coding examples in Chapters 2 and 3 for students who have not programmed in the language. This would provide an introduction to C that should be adequate for the rest of the book.
- Experienced programmers who are using this book to learn x86-64 assembly language on their own should be able to skim the first five chapters. I believe that the remaining chapters would provide a good "primer" for reading the appropriate manuals.

Production of the Book

I used LaTeX 2$\varepsilon$to typeset and draw the figures for this book. The fonts are DejaVu (`dejavu-fonts.org`).

Acknowledgements

I would like to thank the many students who have taken assembly language from me. They have asked many questions that caused me to think about the subject and how I can better explain it. They are the main reason I have written this book.

Three students deserve special thanks, David Tran, Zack Gold, and Jim O'Hara. They used this book in a class taught by Mike Lyle at Santa Rosa Junior College, David in Fall 2010, Zack in Fall 2011, and Jim in Fall 2013. All three caught many of my typos and errors and gave me many helpful suggestions for clarifying my writing. I am very grateful for their careful reading of the book and the time they spent providing me with comments. It is definitely a better book as a result of their diligence.

I wish to thank Richard Gordon, Lynn Stauffer, Allan B. Cruse, Michael Lyle, Suzanne Rivoire, and Tia Watts for their thorough proofreading and critique of the previous versions of this book. By teaching from this book they have caught many of my errors and provided many excellent suggestions for clarifying the presentation.

I appreciate the work of those who volunteer their time to develop and maintain the software I used to create this book: GNU, Linux, LaTeX 2ε, etc.

In addition, I would like to thank my partner, João Barretto, for encouraging me to write this book and putting up with my many hours spent at my computer.

Finally, I am sure there are typos and errors left in this book, even with all the feedback I have received from students and colleagues and my efforts to correct what they found. But I hope it is in good enough shape that you will find reading the book relatively comfortable and that it will provide you some insight into how computers are organized.