A Gentle Introduction to Assembly Language Programming

This textbook provides a gentle introduction to assembly language programming. What makes this introduction "gentle" is that it assumes the reader is already comfortable with C or C++ coding. We use this assumed knowledge to **bridge** backward towards the low level ISA (Instruction Set Architecture).

We drive home a very sharp point:

Assembly language is nothing to be scared of!

For Whom Is This Book Intended?

As mentioned, if you are already familiar with C (or languages descended from C such as C++), this book begins with what you already know. Later chapters dive more deeply into the corners and recesses of the ARM V8 ISA and are suitable for those wishing to master the rich instruction set of the 64 bit ARM processors.

Can This Book Be Used In Courses Covering Assembly Language?

Yes, absolutely.

In fact, we would argue that the study of assembly language is extremely important to the building of competent software engineers. Further, we would argue that teaching the x86 instruction set is cruel as that ISA was born in the 1970s and has simply gotten more muddled with age.

The MIPS instruction set is another ISA that is often covered in College level courses. While kinder and gentler than the x86 ISA, the MIPS processor isn't nearly as relevant as the ARM family. Phones, tablets, laptops and even desktops contain ARM V8 processors making the study of the ARM ISA far more topical. Perhaps even more "cool".

Calling Convention Used In This Book

Assembly language programming is quite closely dependent upon the underlying hardware architecture. The host operating environment plays an outsized role in determining how assembly language programs are constructed. A "calling convention" refers to how functions are called and how parameters are passed.

In this book we will use the ARM LINUX conventions. This means:

- You *may* need to run a ARM Linux VM on the Macintosh - even on ARM-based Macs. Why? Apple uses a different calling convention. Keep reading before you get upset.

The convention used in this book should work on all ARM Linux machines while the Apple calling convention is specific to Apple Silicon-based machines.

This necessity for a VM even when running on an Apple Silicon machine did not sit well with some on reddit. We listened.

We now have a chapter devoted to bringing Linux and Apple code together to the degree possible.

This chapter provides a suite of macros that provide this help. If you're willing to adjust how you code (and use the macros), you can successfully write assembly language once and build it on both Linux and Mac OS.

The macros are a work in progress. This link will lead to a current copy of them. We will try to keep this file up to date.

- You will need to run WSL (Windows Subsystem for Linux) on ARM-based Windows machines. These do exist!
- You will need to run an ARM Linux VM on x86-based Windows machines. This is true even if you are on an ARM-based Windows machine as there are so many differences between a Unix-like environment and Windows.

You'll notice right away that we make use of the C-runtime directly rather than make OS service calls. So, for instance, if we want to call `write()`, we call `write` from the assembly language. This version of the system call `write` is a wrapper function built into the C-runtime (CRT) which handles the lower level details of performing a system call. See the here on what actually happens inside these wrapper functions.

The benefit of using the CRT wrappers is that there are details, explained in the chapter, that differ from system to system and architecture to architecture even for making the same system call.

A Lot of Names

As commendable as the ARM designs are, ARM's naming conventions for their Intellectual Properties are horrid. In this book, AARCH64 and ARM V8 are taken to be synonyms for the 64 bit ARM Instruction Set Architecture (ISA).

It is very difficult to find documentation at the ARM site because they have *so many versions*, so many names for the same thing and so much documentation in general. It really can be maddening.

Within the text we will provide germane links as appropriate.

Here is a link to "a" main instruction set page.

What you need to work with assembly language on Linux

Getting the tools for assembly language development is quite straight forward - perhaps you already have them. Using `apt` from the Linux terminal, say:

```
sudo apt update
sudo apt install build-essential gdb
```

Then you'll need your favorite editor. We currently use `vi` for quick edits and Visual Studio Code for any heavy lifting.

How to build an assembly language

We use `gcc`, the C "compiler". `g++` could also be used. What sense does that make… using the "compiler" to "compile" assembly language?

Well, to answer that one must understand that the word "compiler" refers to only one step in a build sequence. What we talk about as being the "compiler" is actually an umbrella that includes:

- A preprocessor that acts on any `#` preprocessor command like `#include`. These commands are not part of C or C++. Rather they are commands to the preprocessor.

  Note that `gcc` will invoke the C preprocessor only if your assembly language file ends in `.S` - capital S. It may not be invoked if your file ends in a lower case s or any other file extension.

- The *actual* compiler, whose job it is turn high level languages such as C and C++ into assembly language.
- The assembler, which turns assembly language into machine code which is not quite ready for execution.
- And finally, the linker, which combines potentially many intermediate machine code files (called object files), potentially many library files (statically linked .dlls on Windows and .a files on Linux). The linker is the last step in this chain.

Here is a video explaining this process.

We use gcc and g++ directly because, being umbrellas, they automate the above steps with other benefits such as automatically linking in the C runtime.

Suppose you've implemented `main()` in a C file (main.c) and want to call out to an assembly language file you have written (asm.s). It can be done in several ways.

All at once

```
gcc main.c asm.s
```

That's all you need for a minimal build. The resulting program will be written to `a.out`. All the intermediate files generated will be removed.

Modularly

```
gcc -c main.c
gcc -c asm.s
gcc main.o asm.o
```

Used in this way, `.o` files are left on disk. Using the previous method, the `.o` files are removed without you seeing them.

If there are no C or C++ modules used

Suppose `main()` is implemented in assembly language and `main.s` is self-contained, then simply:

```
gcc main.s
```

Often, you will want to enable the debugger `gdb`. Do this:

```
gcc -g main.s
```

The C Pre-Processor

If you want `gcc` to run your code through the C pre-processor (for handing `#include` for example), name your assembly language source code files with a capital S. So, on Linux:

```
gcc main.s
```

Will not go through the C pre-processor but

```
gcc main.S
```

will.

See the Apple Silicon chapter for more information.

Programs called by the "Compiler"

Using gcc to "compile" a program causes the following to be called on Ubuntu running on ARM:

```
/usr/bin/cpp
/usr/lib/gcc/aarch64-linux-gnu/11/cc1
/usr/bin/as
/usr/lib/gcc/aarch64-linux-gnu/11/collect2 which is...
/usr/bin/ld
```

`cpp` is the C preprocessor - it is a general tool can is used by other languages as well (C++, for example).

`cc1` is the actual compiler.

`as` is the assembler.

`ld` is the linker.

You can see why we default to using the umbrella command in this book.

Section 1 - Bridging from C / C++ to Assembly Language

We start by providing what we're calling "bridging" from C and C++ to assembly language. We use the knowledge you already have to learn new knowledge - how cool is that!

Chapter	Markdown	PDF
1	Hello World	Link
2	If Statements	Link
3	Loops	
.... a	.... While Loops	Link
.... b	.... For Loops	Link
.... c	.... Implementing Continue	Link
.... d	.... Implementing Break	Link
4	Interludes	
.... a	.... Registers	Link
.... b	.... Load and Store	Link
.... c	.... More About `ldr`	Link
.... d	.... Register Sizes	Link
5	`switch`	Link
6	Functions	
.... a	.... Calling and Returning	Link
.... b	.... Passing Parameters	Link
.... c	.... Example of calling some common C runtime functions	Link
7	FizzBuzz - a Complete Program	Link
8	Structs	
.... a	.... Alignment	Link
.... b	.... Defining	Link
.... c	.... Using	Link
9	`const`	Link
10	Casting	Link

Section 2 - Floating Point

Floating point operations use their own instructions and their own set of registers. Therefore, floating point operations are covered in their own section:

Chapter	Markdown	PDF
1	Floating Point	
.... a	.... What Are Floating Point Numbers?	Link
.... b	.... Registers (simplified)	Link
.... c	.... Literals	Link
.... d	.... `fmov` Not Yet Written	NA
.... e	.... Conversion To / From Integers	Link
.... f	.... Four Basic Operations Not Yet Written	NA
.... g	.... Selected Additional Operations Not Yet Written	NA
.... z	.... Half Precision Floats	Link

Section 3 - Bit Manipulation

What would a book about assembly language be without bit bashing?

Chapter	Markdown	PDF
1	Bit Fields	
.... a	.... Without Bit Fields	[Link]./section_3/bitfields/README.pdf)
.... b	.... With Bit Fields	Link
.... c	.... Review of Newly Described Instructions	Link
2	Endianness	Link

Section 4 - More Stuff

Chapter	Markdown	PDF
—	Determining string literal lengths for C functions	Link
—	Under the hood: System Calls	Link
—	Apple Silicon	Link

Projects

Here are some project specifications to offer a challenge to your growing mastery. Here are very brief descriptions presented in alphabetical order.

Perhaps before you tackle these, check out the fully described FIZZBUZZ program first.

Then try this as your very first project. With some blank lines and comments it weighs in at 35 lines.

The DIRENT project demonstrates how a complex `struct` can be used in assembly language.

The PI project demonstrates floating point instructions. The program will "throw darts at a target," calculating an approximation of PI by tracking how many darts "hit the target" versus the total number of darts "thrown".

The SNOW project uses 1970's era tech to animate a simple particle system. This project demonstrates a reasonable design process of breaking down complex problems into simpler parts.

The WALKIES presents a cute little animation demonstrating looping with some pointer dereferencing.

About The Author

Perry Kivolowitz's career in the Computer Sciences spans just under five decades. He launched more than 5 companies, mostly relating to hardware, image processing and visual effects (for motion pictures and television). Perry received Emmy recognition for his work on the The Gathering, the pilot episode of Babylon 5. Later he received an Emmy Award for Engineering along with his colleagues at SilhouetteFX, LLC. SilhouetteFX is used in almost every significant motion picture for rotoscoping, paint, tracking, 2D to 3D reconstruction, compositing and more.

In 1996 Perry received an Academy Award for Scientific and Technical Achievement for his invention of Shape Driven Warping and Morphing. This is the technique responsible for many of the famous effects in Forrest Gump, Titanic and Stargate.

Twenty twenty two marks Perry's 18th year teaching Computer Science at the college level, ten years at the UW Madison and now 8 at Carthage College.

Assembly language is a passion for Perry having worked in the following ISAs:

- Univac 1100
- Digital Equipment Corporation PDP-11
- Digital Equipment Corporation VAX-11
- Motorola 68000
- ARM beginning with AARCH64

This work is dedicated to my wife Sara and sons Ian and Evan.

Gratuitous Plugs

Perry has created a library of about 200 programming projects suitable for CS 1, CS 2, Data Structures, Networking, Operating Systems and Computer Organization classes. If a publisher of CS text books be interested in purchasing the library, please reach out.

Also, check out Get Off My L@wn, a Zombie novel for coders. You read that right… elite programmer Doug Handsman retires to his wife Ruth Ann's native northern Wisconsin. And then, well, the apocalypse happens. Bummer.

Rated 4.3 out of 5 with more than 70 reviews, it's a fun read and costs next to nothing.