
HelloSilicon

An introduction to assembly on Apple silicon Macs.

Introduction

In this repository, I will code along with the book *Programming with 64-Bit ARM Assembly Language*, adjusting all sample code for Apple’s ARM64 line of computers. While Apple’s marketing material seems to avoid a name for the platform and talks only about the M1 processor, the developer documentation uses the term “Apple silicon”. I will use this term in the following.

The original source code can be found [here](#).

Prerequisites

While I pretty much assume that people who made it here meet most if not all required prerequisites, it doesn’t hurt to list them.

- You need Xcode 12.2 or later, and to make things easier, the command line tools should be installed. This ensures that the tools are found in default locations (namely `/usr/bin`). If you are not sure that the tools are installed, check *Preferences* → *Locations* in Xcode or run `xcode-select --install`.
- All application samples also require at least macOS Big Sur, iOS 14 or their respective watchOS or tvOS equivalents. Especially for the later three systems it is not a necessity per-se (neither is Xcode 12.2), but it makes things a lot simpler.
- Finally, while all samples can be adjusted to work on the iPhone and all other of Apple’s ARM64 devices, for best results you should have access to an Apple silicon Mac.

Acknowledgments

I would like to thank @claudi, @jannau, @jrosengarden, @m-schmidt, @saagarjha, and @zhuowei! They helped me when I hit a wall, or asked questions that let me improve the content.

Changes To The Book

With the exception of the existing iOS samples, the book is based on the Linux operating system. Apple’s operating systems (macOS, iOS, watchOS and tvOS) are actually just flavors of the Darwin operating system, so they share a set of common core components.

Linux and Darwin, which were both inspired by AT&T Unix System V, are significantly different at the level we are looking at. For the listings in the book, this mostly concerns system calls (i.e. when we want the Kernel to do something for us), and the way Darwin accesses memory.

This file is organized so that you can read the book, and read about the differences for Apple silicon side by side. The headlines in this document follow those in the book.

Chapter 1: Getting Started

Computers and Numbers

The default Calculator.app on macOS has a “Programmer Mode”, too. You enable it with *View* → *Programmer* (⌘3).

CPU Registers

Apple has made certain platform specific choices for the registers:

- Apple reserves **X18** for its own use. Do not use this register.
- The frame pointer register (**FP**, **X29**) must always address a valid frame record.

About the GCC Assembler

The book uses Linux GNU tools, such as the GNU `as` assembler. While there is an `as` command on macOS, it will invoke the integrated LLVM Clang assembler by default. And even if there is the `-Q` option to use the GNU based assembler, this was only ever an option for `x86_64` — and is already deprecated as of this writing.

```
1 % as -Q -arch arm64
2 /usr/bin/as: can't specify -Q with -arch arm64
```

Thus, the GNU assembler syntax is not an option, and the code will have to be adjusted for the Clang assembler syntax.

Likewise, while there is a `gcc` command on macOS, this simply calls the Clang C-compiler. For transparency, all calls to `gcc` will be replaced with `clang`.

```
1 % gcc --version
2 Configured with: --prefix=/Applications/Xcode-beta.app/Contents/
  Developer/usr --with-gxx-include-dir=/Applications/Xcode-beta.app/
  Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.
  sdk/usr/include/c++/4.2.1
```

```
3 Apple clang version 12.0.0 (clang-1200.0.32.27)
4 Target: arm64-apple-darwin20.1.0
5 Thread model: posix
6 InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/
  XcodeDefault.xctoolchain/usr/bin
```

Hello World

If you are reading this, I assume you already knew that the macOS Terminal can be found in *Applications → Utilities → Terminal.app*. But if you didn't I feel honored to tell you and I wish you lots of fun on this journey! Don't be afraid to ask questions.

To make "Hello World" run on Apple silicon, first the changes from page 78 (Chapter 3) have to be applied to account for the differences between Darwin and the Linux kernel. To silence the warning, I insert `.align 4` (or `.p2align 2`), because Darwin likes things to be aligned on even boundaries. The book mentions this in *Aligning Data* in Chapter 5, page 114.

System calls in Linux and macOS have several differences due to the unique conventions of each system. Here are some key distinctions:

- * **Function Number:** The function numbers differ between the two systems, with Linux using 64 and macOS using 4. The table for Darwin (Apple) system calls can be found at this link: [Darwin System Calls](#). Please note that this is a specific version (the most recent at the time of writing), and newer versions can be found [here](#). > [!CAUTION] > Darwin function numbers are considered private by Apple, and are subject to change. They are provided here for educational purposes only
- * **Address for Storing Function Numbers:** The address used for storing function numbers also varies. In Linux, it's on X8, while in macOS, it's on X16.
- * **Interrupt Call:** The call for interruption is 0 in Linux, whereas it's 0x80 on Apple Silicon.

To make the linker work, a little more is needed, most of it should look familiar to Mac/iOS developers. These changes need to be applied to the `makefile` and to the `build` file. The complete call to the linker looks like this:

```
1 ld -o HelloWorld HelloWorld.o \
2   -lSystem \
3   -syslibroot `xcrun -sdk macosx --show-sdk-path` \
4   -e _start \
5   -arch arm64
```

We know the `-o` switch, let's examine the others:

- `-lSystem` tells the linker to link our executable with `libSystem.dylib`. We do that to add the `LC_MAIN` load command to the executable. Generally, Darwin does not support statically linked executables. It is possible, if not especially elegant to create executables without using

`libSystem.dylib`. I will go deeper into that topic when time permits. For people who read *Mac OS X Internals* I will just add that this replaced `LC_UNIXTHREAD` as of MacOS X 10.7.

- `-sysroot`: In order to find `libSystem.dylib`, it is mandatory to tell our linker where to find it. It seems this was not necessary on macOS 10.15 because “*New in macOS Big Sur 11 beta, the system ships with a built-in dynamic linker cache of all system-provided libraries. As part of this change, copies of dynamic libraries are no longer present on the filesystem.*”. We use `xcrun -sdk macosx --show-sdk-path` to dynamically use the currently active version of Xcode.
- `-e _start`: Darwin expects an entrypoint `_main`. In order to keep the sample both as close as possible to the book, and to allow its use within the C-Sample from *Chapter 3*, I opted to keep `_start` and tell the linker that this is the entry point we want to use
- `-arch arm64` for good measure, let’s throw in the option to cross-compile this from an Intel Mac. You can leave this off when running on Apple silicon.

Reverse Engineering Our Program

While the `objdump` command line program works just as well on Darwin and produces the expected output, also try the `--macho` (or `-m`) option, which causes `objdump` to use the Mach-O specific object file parser.

Chapter 2: Loading and Adding

The changes from Chapter 1 (makefile, alignment, system calls) have to be applied.

Register and Shift

The gcc assembler accepts `MOV X1, X2, LSL #1`, which is not defined by the ARM Compiler User Guide, instead `LSL X1, X2, #1` (etc) is used. After all, both are just aliases for the instruction `ORR X1, XZR, X2, LSL #1`.

Register and Extension

Clang requires the source register to be 32-bit. This makes sense because with these extensions, the upper 32 Bit of a 64-bit register will never be touched:

```
1 ADD X2, X1, W0, SXTB
```

The GNU Assembler seems to ignore this and allows you to specify a 64-bit source register.

Chapter 3: Tooling Up

Beginning GDB

On macOS, `gdb` has been replaced with the LLDB Debugger `lldb` of the LLVM project. The syntax is not always the same as for `gdb`, so I will note the differences here.

To start debugging our **movexamps** program, enter the command

```
1 lldb movexamps
```

This yields the abbreviated output:

```
1 (lldb) target create "movexamps"
2 Current executable set to 'movexamps' (arm64).
3 (lldb)
```

Commands like `run` or `list` work just the same, and there is a nice GDB to LLDB command map.

To disassemble our program, a slightly different syntax is used for `lldb`:

```
1 disassemble --name start
```

Note that because we are linking a dynamic executable, the listing will be long and include other `start` functions. Our code will be listed under the line `movexamps`start`.

Likewise, `lldb` wants the breakpoint name without the underscore: `b start`

To get the registers on `lldb`, we use **register read** (or `re r`). Without arguments, this command will print all registers, or you can specify just the registers you would like to see, like `re r SP X0 X1`.

We can see all the breakpoints with **breakpoint list** (or `br l`). We can delete a breakpoint with **breakpoint delete** (or `br de`) specifying the breakpoint number to delete.

`lldb` has even more powerful mechanisms to display memory. The main command is **memory read** (or `m read`). For starters, these are the parameters used by the book:

```
1 memory read -fx -c4 -s4 $address
```

where `* -f` is the display format `* -s` size of the data `* -c` count

Listing 3-1

As an exercise, I have added code to find the default Xcode toolchain on macOS. In the book they are using this to later switch from a Linux to an Android toolchain. This process is much different

for macOS and iOS: It does not usually involve a different toolchain, but instead a different Software Development Kit (SDK). You can see this in Listing 1-1 where `-sysroot` is set.

That said, while it is possible to build an iOS executable with the command line it is not a trivial process. So for building apps I will stick to Xcode.

Apple Xcode

As Chapter 10 focusses on building an app that will run on iOS, I have chosen to simply create a Command Line Tool here which is now using the same `HelloWorld.s` file.

Be aware that the function numbers are not only different, but on Darwin, they are considered private and subject to change.

Chapter 4: Controlling Program Flow

Besides the common changes, we face a new issue which is described in the book in Chapter 5: Darwin does not like `LDR X1, =symbol`, it will produce the error `ld: Absolute addressing not allowed in arm64 code`. If we use `ASR X1, symbol`, as suggested in Chapter 3 of the book, our data has to be in the read-only `.text` section. In this sample however, we want writable data.

The Apple Documentation tells us that on Darwin: > All large or possibly nonlocal data is accessed indirectly through a global offset table (GOT) entry. The GOT entry is accessed directly using RIP-relative addressing.

And by default, on Darwin all data contained in the `.data` section, where data is writeable, is “possibly nonlocal”.

The full answer can be found here: > The `ADRP` instruction loads the address of the 4KB page anywhere in the +/-4GB (33 bits) range of the current instruction (which takes 21 high bits of the offset). This is denoted by the `@PAGE` operator. then, we can either use `LDR` or `STR` to read or write any address inside that page or `ADD` to calculate the final address using the remaining 12 bits of the offset (denoted by `@PAGEOFF`).

So this:

```
1    LDR X1, =outstr // address of output string
```

becomes this:

```
1    ADRP    X1, outstr@PAGE // address of output string 4k page
2    ADD X1, X1, outstr@PAGEOFF // offset to outstr within the page
```

Exercises

I was asked how to read the command line, and I gladly answered the question.

Sample code can be found in Chapter 4 in the file `case.s`.

Chapter 5: Thanks for the Memories

The important differences in memory addressing for Darwin were already addressed above.

Listing 5-1

The `quad`, `octa` and `fill` keywords must be in lowercase for the llvm assembler. (See bottom of this file)

Listing 5-10

Changes like in Chapter 4.

Chapter 6: Functions and the Stack

As we learned in Chapter 5, all assembler directives (like `.equ`) must be in lowercase.

Chapter 7: Linux Operating System Services

`asm/unistd.h` does not exist in the Apple SDKs, instead `sys/syscalls.h` can be used.

Warning: Be aware that syscall numbers in Darwin are officially considered private and subject to change. They are presented here for educational purposes only.

It is also important to notice that while the calls and definitions look similar, Linux and Darwin are not the same: `AT_FDCWD` is -100 on Linux, but must be -2 on Darwin.

Unlike Linux, errors are signified by setting the carry flag, and the error codes are non-negative. We therefore `MOV` the result into the required register instead of `ADD` (we don't need to check for negative numbers, and need to preserve the condition flags) and `B.CC` to the success path.

Chapter 8: Programming GPIO Pins

This chapter is specifically for the Raspberry Pi 4, so there is nothing to do here.

Chapter 9: Interacting with C and Python

For transparency reasons, I replaced `gcc` with `clang`.

Listing 9-1

Apart from the usual changes, Apple diverges from the ARM64 standard ABI (i.e. the convention how functions are called) for variadic functions. Variadic functions are functions which take a variable number of arguments, and `printf` is one of them. Where Linux will accept arguments passed in the registers we must pass them on the stack for Darwin.

```
1 str    X1, [SP, #-32]! // Move the stack pointer four doublewords (32
    bytes) down and push X1 onto the stack
2 str    X2, [SP, #8]    // Push X2 to one doubleword above the current
    stack pointer
3 str    X3, [SP, #16]   // Push X3 to two doublewords above the current
    stack pointer
4 adrp   X0, ptfStr@PAGE // printf format str
5 add    X0, X0, ptfStr@PAGEOFF // add offset for format str
6 bl     _printf // call printf
7 add    SP, SP, #32 // Clean up stack
```

So first, we are growing the stack downwards 32 bytes to make room for three 64-bit values. We are creating space for a fourth value for padding because, as pointed out on page 137 in the book, ARM hardware requires the stack pointer to always be 16-byte aligned.

In the same command, **X1** is stored at the new location of the stack pointer.

Now, we fill the rest of the space that was just created by storing **X2** in a location eight bytes above, and **X3** 16 bytes above the stack pointer. Note that the **str** commands for **X2** and **X3** do not move **SP**.

We could fill the stack in different ways; what is important that the `printf` function expects the parameters as doubleword values in order, upwards from the current stackpointer. So in the case of the `debug.s` file, it expects the parameter for the `%c` to be at the location of **SP**, the parameter for `%32ld` at one doubleword above this, and finally the parameter for `%016lx` two doublewords, 16 bytes, above the current stack pointer.

What we have effectively done is allocating memory on the stack. As we, the caller, “own” that memory we need to release it after the function branch, in this case simply by shrinking the stack (upwards) by the 32 bytes we allocated. The instruction `add SP, SP, #32` will do that.

Listing 9-5

`mytoupper` was prefixed with `_` as this is necessary for C on Darwin to find it.

Listing 9-6

No change was required.

Listing 9-7

Instead of a shared `.so` ELF library, a dynamic Mach-O library is created. Further information can be found here: [Creating Dynamic Libraries](#)

Listing 9-8

In inline-assembly, which we are using here, The `cont` label must be declared as a local label by prefixing it with `L`. While this was not necessary in pure assembly, like in Chapter 5, the LLVM C-Frontend will automatically add the directive `.subsections_via_symbols` to the code:

Funny Darwin hack: This flag tells the linker that no global symbols contain code that falls through to other global symbols (e.g. the obvious implementation of multiple entry points). If this doesn't occur, the linker can safely perform dead code stripping. Since LLVM never generates code that does this, it is always safe to set. (From LLVM source code)

While we are using the LLVM toolchain, in assembly — including inline-assembly — all safety checks are off so we must take extra precautions and specifically declare the forward label local.

Also, the size of one variable had to be changed from `int` to `long` to make the compiler complete happy and remove all warnings

Calling Assembly from Python

Listing 9-9

While the `uppertst5.py` file only needed a minimal change, calling the code is a little more challenging. On Apple silicon Macs, Python is a Mach-O universal binary with two architectures, `x86_64` and `arm64e`:

```
1 % lipo -info /usr/bin/python3
2 Architectures in the fat file: /usr/bin/python3 are: x86_64 arm64e
```

Notably absent is the arm64 architecture we were building for up to this point. This makes our dylib unusable with Python.

arm64e is the Armv-8 architecture, which Apple is using since the A12 chip. If you want to address devices prior to the A12, you must stick to arm64. The first Macs to use ARM64 run on the M1 CPU based on the A14 architecture, thus Apple decided to take advantage of the new features.

So, what to do? We could compile everything as arm64e, but that would make the library useless on devices like the iPhone X or older, and we would like to support them, too.

Above, you read something about a *universal binary*. For a very long time, the Mach-O executable format was supporting several processor architectures in a single file. This includes, but is not limited to, Motorola 68k (on NeXT computers), PowerPC, Intel x86, as well ARM code, each with their 32 and 64 bit variants where applicable. In this case, I am building a universal dynamic library which includes both arm64 and arm64e code. More information can be found [here](#).

While most of the Python IDEs that work for Linux are also available for macOS, as of this writing, the only Python IDEs which itself runs as arm64 — and thus will load arm64 libraries — is Python.org IDLE, version 3.10 or newer.



Figure 9-1. Our Python program running in the IDLE IDE

Alternatively, you can use the command line to test the program. (As of macOS 12.3, Apple removed Python 2 and developers should use Python 3)

```
1 % python3 uppertst5.py
2 b'This is a test!'
3 b'THIS IS A TEST!'
4 16
```

A final note: While the Apple python3 binary is arm64e, the Python Framework used by IDLE is arm64. The fact that the library built in this chapter is a universal binary containing both architectures allows it to be used in either environment.

Chapter 10: Interfacing with Kotlin and Swift

No changes in the core code were required, but instead of just an iOS app I created a SwiftUI app that will work on macOS, iOS, watchOS (Series 4 and later), and tvOS.

Chapter 11: Multiply, Divide, and Accumulate

At this point, the changes should be self-explanatory. The usual makefile adjustments, `.align 4`, address mode changes, and `_printf` adjustments.

Chapter 12: Floating-Point Operations

Like in Chapter 11, all the changes have been introduced already. Nothing new here.

Chapter 13: Neon Coprocessor

Once again, the Clang assembler wants a slightly different syntax: Where gcc accepts

```
1 MUL V6.4H, V0.4H, V3.4H[0]
```

the Clang assembler expects

```
1 MUL.4H V6, V0, V3[0]
```

All other changes to the code should be trivial at this point.

Chapter 14: Optimizing Code

No unusual changes here.

Chapter 15: Reading and Understanding Code

Copying a Page of Memory

Some place to start reading ARM64 code in the Darwin Kernel can be found in `bcopy.s`. There is a lot more in that directory and the repository in general.

Code Created by GCC

No changes were required. The “tiny” code model is not supported for Mach-O executables:

```
1 % clang -O3 -mmodel=tiny -o upper upper.c
2 fatal error: error in backend: tiny code model is only supported on ELF
```

Chapter 16: Hacking Code

All that can be said is that clang automatically enables position-independent executables, and the option `-no-pie` does not work. Therefore, the exploit shown in the `upper.s` file can not be reproduced.

Additional references

- Writing ARM64 Code for Apple Platforms, documentation how Apple platforms diverge from the standard 64-bit ARM architecture
- Mach-O Programming Topics, an excellent introduction to the Mach-O executable format and how it differs from ELF. Even if it still references PowerPC 64-bit architecture and says nothing about ARM, most of it is still true.
- What is required for a Mach-O executable to load?
- Mac OS X Internals, A Systems Approach Amit Singh, 2007. For better or worse, this is still the definite compendium on the core of macOS and it’s siblings.
- WWDC20: Explore the new system architecture of Apple silicon Macs A system overview of the new Apple silicon machines
- Darwin Source Code
- ARM Architecture Reference Manual

One More Thing...

“The C language is case-sensitive. Compilers are case-sensitive. The Unix command line, ufs, and nfs file systems are case-sensitive. I’m case-sensitive too, especially about product names. The IDE is called Xcode. Big X, little c. Not XCode or xCode or X-Code. Remember that now.” — Chris Espinosa