# Apple Silicon

This book is written to the Linux calling convention as stated early on. Unfortunately, this means that even if you own an Apple Silicon machine, which is AARCH64, you'd still need a Linux virtual machine.

This didn't sit well with some on reddit and rightfully so. We undertook to develop a way of writing assembly code once and having it work on both Mac OS and Linux to the degree possible.

Much of this chapter has been replaced here.

There are some things we cannot adapt, such as variadic functions (e.g. `printf()`) but we explain how code can be written to be compatible with both environments at the expense of some minor amount duplicated code.

## Assembly language macros

An early innovation in assemblers was the introduction of a macro capability. Given what could be considered a certain amount of tedium in coding in asm, macros provide a simple form of *meta programming* where a series of statements can be encapsulated by a single macro. Think of a macro as an early form of C++ templated function (kinda but not really).

Here's an example of an assembly language macro:

```
.macro LLD_ADDR xreg, label
        adrp    \xreg, \label@PAGE
        add     \xreg, \xreg, \label@PAGEOFF
.endm
```

Here's how it might be used:

```
        LLD_ADDR x0, fmt
```

This gets expanded to:

```
        adrp    x0, fmt@PAGE
        add     x0, x0, fmt@PAGEOFF
```

## Reminder - there is documentation for these macros

The documentation for the macro suite has been moved here.

## How to force the C pre-processor to run on assembly language

`clang` on Mac OS will run assembly language files through the C pre-processor. `clang` on Linux will not by default but can if you specify `-x assembler-with-cpp`.

`gcc` on Mac OS can be based on `clang` so on Mac OS it inherits `clang`'s behavior. `gcc` on Linux does not run assembly language files through the C pre-processor *if the asm file ends in .s but WILL if the file ends in .S*

## Differences between Apple and Linux

### Getting the address of `errno`

`errno` is an externally defined `int32_t` used by many "system" provided APIs to report error conditions back to calling programs. The macro `ERRNO_ADDR` can be used to converge how Linux and Apple get the address of the variable (which is left in `x0`).

### Variadic functions

*This is important! Understand this section in order to be able to use `printf()`.*

Functions like `printf()` are variadic. These are functions that can take any number of parameters. The first argument contains information that tells the function how many parameters were actually given.

For example:

```
printf("%d is a number.\n", 9);
```

There is but one `%` place holder in this text. This tells `printf()` that in addition to the string there is but one more parameter to be expected.

Apple and Linux handle variadic differently.

Linux will use the scratch registers first up to the integer or floating point register 7. *Then* it will use the stack.

Apple will put the first parameter in the zero register and then shifts immediately to putting all other parameters onto the stack.

We overcome this difference by detecting which environment we are building in using `#if` after having first set up for the Linux version.

By setting up for the Linux version, the Apple version involves just pushing registers onto the stack.

Remember that **`%f` always** expects a double. This is hidden from you in C and C++ but is important in assembly language. Use `fcvt` to shift from single precision to double.

An example:

```
        LLD_ADDR    x0, fmt
        LLD_FLT     x1, s0, flt
        fcvt        d0, s0
#if defined(__APPLE__)
```

```
        PUSH_R      d0
        CRT         printf
        add         sp, sp, 16  // See discussion below.
#else
        CRT         printf
#endif
```

Reminder that this makes use of the C preprocessor to perform the conditional evaluation detecting the platform. You can ensure that the C preprocessor is used by naming your assembly language source code files ending in capital S.

### Undoing Stack Pointer Changes

A small tip concerning undoing changes to the stack pointer. You might think that changes to the stack made by `str` or `stp` and their cousins **must** be undone with `ldr` or `ldp` and their cousins.

This depends.

If you need to get back the original contents of a register pushed onto the stack, then an `ldr` or `ldp` is appropriate. However, if you don't need to get the original contents of a register back, then it is faster to undo a change to the stack using addition.

Take for example the use of `printf()`. On Apple Silicon systems, you must send arguments to `printf()` by pushing them onto the stack. However, when `printf()` completes, you have no need for the values that you pushed. As shown above, simply add the right (multiple of 16) to the stack pointer. This is faster as the addition makes no reference to RAM (or caches) as the `ldr` would.

### Frame pointer

Apple requires that `x29` be kept as a valid stack frame pointer. The frame pointer should always start out as equal to the stack pointer. However, within the function, the stack pointer is free to change. The frame pointer must remain fixed so that debuggers always know how to find the initial stack *frame*.

To be Apple compatible, in addition to backing up `x30` also back up `x29` and then:

```
mov x29, sp
```

We will be converting all sample code to do this over time.

### More?

As we discover more differences, they will be described here.

## START_PROC and END_PROC

Again, for debugging purposes, you can insert frame checks into your code. These work the same on both Apple Silicon and Linux. If you want these, put `START_PROC` after the label introducing a function. Then, put `END_PROC` after the last statement of the function.

This helps the debuggers understand where a function begins and ends.

We will transition all sample code to use this over time.

## A useful link

Here is an understandable version of gcc documentation.