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.

We are pleased to present this chapter along with a set of assembly language macros that, if used, help a great deal.

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 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
```

Loading the address of data

Assuming:

```
        .data
fmt:    .asciz   "Hello!"
```

When we:

```
ldr x0, =fmt
```

we are hoping to put the address of the label `fmt` into `x0`. But how would this be possible since we've seen that addresses are (often) six bytes long and our instructions are always 4 bytes long? As we describe elsewhere, the above `ldr` instance is actually turned into instructions to load an address relative to the address of the current instruction.

As long as the data we want is relatively close to the `ldr`, this works out to a difference in addresses that is small (and so, can be fit into a 4 byte instruction).

Apple does not allow instructions of the form:

```
ldr x0, =fmt
```

Instead they take a more general approach of splitting addresses of data into two parts:

1. The *page* on which the label lives - think of this as generating the upper bits of the address.
2. The *offset* on the page where the label actually resides - think of this as the lower bits of the address.

Hence:

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

The first instruction puts the high bits of the label's address in `x0`. Then, the second instruction literally adds the low bits of the label's address into `x0` forming a complete address.

In this way, labels can be further away from the current instruction than the Linux way.

Apple does something similar with global variables, perhaps defined in C or C++ files. Instead of `PAGE` and `PAGEOFF` they use global versions. The macro `GLD_ADDR` is used in this case rather than `LLD_ADDR` which works with "locally" defined addresses.

How does this help bridge Apple and Linux?

Here is an assembly language file containing the macros we're developing to bring Linux and Apple Silicon assembly language closer together.

Notice it has:

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

but also:

```
.macro LLD_ADDR xreg, label
        ldr     \xreg, =\label
.endm
```

Which of these are used is determined by whether or not you are assembling on an Apple machine or a Linux machine using features provided by the standard C pre-processor. I.e.:

```
# if defined(__APPLE__)
// apple stuff
# else
// not apple stuff
# endif
```

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

Loading label addresses

This was described above. If you use `LLD_ADDR` the macros will adapt for you.

Function labels

Apple prepends an underscore, Linux does not. Instead of:

```
bl  printf
```

do:

```
CRT printf
```

and the macro will adapt.

main

Like other function labels, Apple wants `_main` while Linux wants `main`.

Simply use:

```
MAIN
```

and the macro will adapt.

Globals

Instead of writing:

`.global main`

use

`GLABEL main`

and the macros will adapt.

You can find documentation on the macros here.

Variadic functions

Functions like `printf()` are variadic. This means the function can take any number of parameters. The first argument contains some 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 `x7`. *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 to print a float or double, they must be copied to `x` registers.

An example:

```
        LLD_ADDR x0, fmt     // loads the address of fmt
        LLD_PTR  x1, ptr     // loads **ptr
        ldr      x1, [x1]    // turns **ptr into *ptr
        ldr      x2, [x1]    // dereferences *ptr to get value
# if defined(__APPLE__)
        // if apple, push the second and third argument to stack
        PUSH_P  x1, x2
        CRT     printf
        add     sp, sp, 16
# else
        // if not apple, the registers are already set up
        CRT     printf
```

```
# endif
```

Other differences

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.