THE ECLECTIC LIGHT COMPANY

MACS, PAINTING, AND MORE

hoakley / June 16, 2021 / Macs, Technology

Code in ARM Assembly: Registers explained

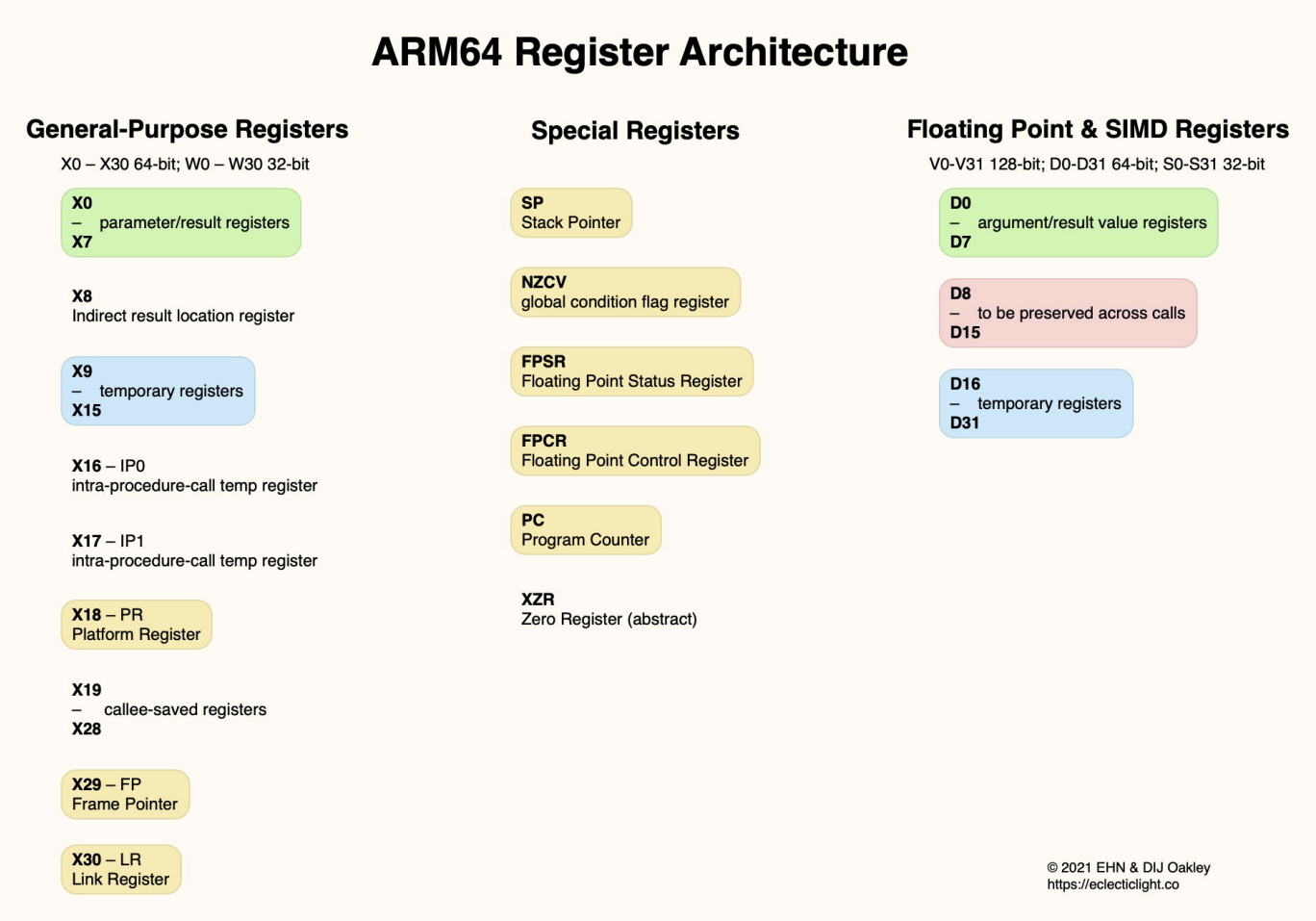
In the first article in this series on developing for Apple Silicon Macs using assembly language, I built a simple framework AsmAttic to use as the basis for developing ARM assembly language routines. In that, I provided a short and simple demonstration of calling an assembly routine and getting its result. This article starts to explain the mechanics of writing your own routines, by explaining the register architecture of ARM64 processors.

In that first article, I glibly produced a C wrapper of
`extern double multadd(double, double, double)`
to call an assembly language routine of

```
_multadd:
STR LR, [SP, #-16]!
FMADD D0, D0, D1, D2
LDR LR, [SP], #16
RET
```

Stepping through the lines of assembly, that first saves a set of registers, performs the floating point operation I want, restores the registers, and returns. For any of that to make sense, you first need to understand the register architecture of the ARM64 processor.

The processor has three main types of register: general-purpose, floating point (including SIMD) and special. When calling and running routines like this, you're most concerned with the first two, which are explained in detail in ARM's Procedure Call Standard (references below), and Apple's platform-specific document (references).



(Tear-out PDF version: armregisterarch)

There are 31 general-purpose registers, each of which can be used for 64- or 32-bit values. When used for 64-bit, they're named X0 to X30, and for 32-bit they're W0 to W30.

There are 32 floating point registers, each of which can be used for 128-bit values as V0 to V31, 64-bit as D0 to D31, and 32-bit as S0 to S31.

For the sake of simplicity, in this series I'll default to using 64-bit values for integers and floating point wherever possible. This means that the registers I'm going to work with most commonly are the X and D series, as in the example above. In higher language equivalents, they equate to data of types `C long`, pointer, Swift Int, `C double` and Swift Double.

The first eight registers in each of these two types, X0-X7 and D0-D7, are used to pass arguments to assembly functions, and the first, X0 and D0, are used to return the result of an assembly routine. They are used in order: the first non-floating point argument will be passed in X0, the second in X1, and so on; the first floating point argument will be passed in D0, the second in D1, and so on.

It's important to remember that these are used according to the register group. For a C wrapper of

```
extern double burble(long, double, long, double)
```

the first long will be passed in X0, the first double in D0, the second long in X1, and the second double in D1, with the result being returned in D0, as it's a double.

This is the simplest way of passing arguments to an assembly routine, in which the *value* is used, and is placed in the register ready for the routine to access: '**call by value**' (or 'pass by value'). The only result returned, in either X0 or D0, is a single value too.

Passing arrays, structures and other arguments which can't simply be put into a single register requires a different method, in which a *pointer* to the data is passed: '**call by reference**'. This is also used to enable a routine to return more than one result: when an argument is passed as a pointer to a floating point number, for example, if the routine changes the value referenced by that pointer, then that changed value is available to the code which calls that routine.

Let's suppose that a floating point routine takes three doubles representing x, y and z co-ordinates, and I want it to return three changed doubles representing those co-ordinates transformed into a projected space. As only one floating point number can be returned as a result, in register D0, what we could do instead is pass those co-ordinates by reference,

```
extern void transform(*double, *double, *double)
```

In Swift, we might call that in turn as

```
transform(&x, &y, &z)
```

to pass the addresses of those Doubles.

In an assembly routine, calling by value is simplest, as the values are available direct from the designated registers. Calling by reference is a bit more complicated, though, as before the routine can access values it first has to load them from the address passed. And that's where my next article starts.

References

Code in Assembly for Apple Silicon with the AsmAttic app (previous article on this blog)
Procedure Call Standard for the Arm 64-bit Architecture (ARM) from Github
Writing ARM64 Code for Apple Platforms (Apple)
Stephen Smith (2020) *Programming with 64-Bit ARM Assembly Language,* Apress, ISBN 978 1 4842 5880 4.
Daniel Kusswurm (2020) *Modern Arm Assembly Language Programming,* Apress, ISBN 978 1 4842 6266 5.
ARM64 Instruction Set Reference (ARM).

Posted in Macs, Technology and tagged Apple silicon, ARM, assembler, assembly language, M1, Xcode. Bookmark the permalink.

Quick Links

Free Software Menu