The ARM processor (Thumb-2), part 16: The calling convention

devblogs.microsoft.com/oldnewthing/20210621-00

June 21, 2021

Raymond Chen

For non-variadic functions, the Windows calling convention for ARM matches the Procedure Call Standard for the Arm Architecture, so this will largely match what you see on most other operating systems.

The fine points of the calling convention are spelled out in the standard document, but here's the short and simple version.

First, parameters smaller than 32 bits are extended to 32-bit values in a manner consistent with their type: Signed types are sign-extended and unsigned types are zero-extended.

Next, parameters are laid out as if inside a giant structure:

```
struct Parameters
{
#if function returns a structure larger than 64 bits
    Ret* result;
#endif
#if function has a "this" pointer
    T* this;
#endif
    T1 param1;
    T2 param2;
    T3 param3;
    T4 param4;
    T5 param5;
    ... etc ...
};
```

Padding is inserted as necessary to preserve natural alignment.

The first 4 × 4 = 16 bytes of the resulting structure are loaded into registers *r0* through *r3*, and the rest are put onto the stack.

For example, suppose we have this function:

```
void f(int8_t a, int64_t b, int16_t c);
```

The parameter `a` is sign-extended to `int32_t` and placed in *r0*.

The next parameter requires 8-byte alignment, so we skip *r1* so that we can reach 8-byte alignment starting at *r2*.

Parameter `b` goes into registers *r2* and *r3*.

Parameter `c` goes onto the stack.

On function return, register *r0* contains the integer value. If the return value is smaller than 32 bits, it is extended to a 32-bit value in a manner consistent with its type. If the return value is a 64-bit integer, then *r1* holds the upper 32 bits of the result. If the return type is a floating point variable, it is returned in *s0* or *d0*, as appropriate.

Things get messier once you introduce floating point parameters.

Each floating point parameter goes into the next available *s#* or *d#* register, starting with *s0* and *d0*. Remember that the *s#* and *d#* registers overlap, so if you use *s0* and then need a double-precision register, you have to move up to *d1*.

On the other hand, parameters can backfill: If you need a single-precision register, you can use an odd-numbered *s#* register that had previously been skipped in order to get to an earlier double-precision parameter.

Floating point registers used for passing parameters are *s0* through *s15*, and *d0* through *d7*. If you run out of floating point registers, then that parameter and all subsequent floating point parameters go onto the stack. (No backfilling after spilling.)

```
void f(int i1, float f1, int i2, double d1, float f2);
```

The parameter assignment for this function goes like this:

- `i1` goes into *r0*.
- `f1` goes into *s0*.
- `i2` goes into *r1*.
- `d1` goes into *d1*. Skip over *s1*.
- `f2` goes into *s1*. Go *backward* to backfill *s1*.

Integer parameters and floating point parameters are allocated independently, and floating point parameters can backfill. Together, this means that a lot of distinct function signatures end up using the same registers:

```
void f(int i1, float f1, int i2, double d1, float f2);
void f(int i1, int i2, float f1, double d1, float f2);
void f(int i1, int i2, float f1, float f2, double d1);
void f(float f1, float f2, double d1, int i1, int i2);
```

All of these functions pass the parameters in the same registers, even though they are listed differently in the source code.

There is no parameter home space on the stack. At function entry, the first stack-based parameter is stored directly at the top of the stack.

If any parameters were put onto the stack, they are the responsibility of the caller to clean up. In practice, instead of cleaning the stack after every call, the caller preadjusts the stack pointer at function entry to reserve space for all outbound stack-based parameters and just reuses the space for each function call, doing the cleanup at the end of the function.

Variadic functions follow a different set of register assignment rules: All floating point parameters are passed as if they were integer parameters: A single-precision floating point parameter is passed as if it were a 32-bit integer, and a double-precision floating point parameter is passed as if it were a 64-bit integer. This rule applies even to the non-variadic parameters.

Next time, we'll look at how these parameter passing rules are implemented in code.

Raymond Chen

Follow