The AArch64 processor (aka arm64), part 20: The classic calling convention

devblogs.microsoft.com/oldnewthing/20220823-00

August 23, 2022

Raymond Chen

For AArch64, Windows employs two calling conventions. One is for classic 64-bit ARM code and the other (named ARM64EC) is for 64-bit ARM code that is intended to interoperate with x64 emulation. The EC stands for "Emulation Compatible". We'll look at the ARM64EC calling convention later.

The classic calling convention follows the Procedure Call Standard for the ARM 64-bit Architecture. This is a rather lengthy document, summarized on MSDN, and which I will further simplify here. If you need to dig into the details for weirdo edge cases, go to the other documents.

Integer and pointer parameters are passed in *x0* through *x7*, and floating point parameters go into *v0* through *v7*.

If a parameter does not fill its assigned register or memory, then the value goes into the low-order bits and the upper bits are *uninitialized*.

If a parameter is a large structure (larger than 16 bytes), then it is passed by address. Small structures are passed by value, packed into integer registers. (Execption: If the small structure consists entirely of floats or entirely of doubles, then it is passed in floating point registers, one for each member.)

Each parameter is assigned the next available register for its value class (integer/pointer or floating point). Since the value goes into the low-order bits, this means that floats are effectively passed in *s#* and doubles in *d#*.

If a parameter is a 128-bit integer, it consumes an even/odd register pair. This may force an odd-numbered integer register to be skipped.

There is no backfilling, as occurred on AArch32. If a floating point register is used to hold a float, the remaining bits are left unused and cannot be used to hold a later float parameter.

If you run out of registers for a particular value class, future parameters go onto the stack. However, parameters of the other value class can still go into registers if registers are still available.

If a structure does not fit entirely in registers, then it goes completely on the stack.

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.

The return value is placed in *x0*/*x1* or *v0*/*v1*, depending on its value class. Again, if the return value doesn't fill the output register, the unused bits are left uninitialized. If the return value doesn't fit in two registers, then the caller passes as a secret first parameter a pointer to a block of memory that receives the return value.

All stack parameters are caller-clean. In practice, instead of cleaning the stack after every call, the caller preadjusts the stack pointer in its prologue to reserve space for all outbound stack-based parameters and just reuses the space for each function call, doing the cleanup in the epilogue.

Here are some examples:

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

    ; x0[ 7:0] = a
    ; x1[63:0] = b
    ; x2[15:0] = c
```

The parameters that are smaller than a 64-bit register occupy the low-order bits of the 64-bit register, and the upper bits are garbage. The recipient may not assume that the upper bits are the zero-extension or sign-extension of the formal parameter.

```
void f(float a, int b, double c, float d)

    ; s0       = a
    ; x0[31:0] = b
    ; d1       = c
    ; s2       = d
```

Notice that parameter *b* goes into *x0* since it is the first integer/pointer parameter. The fact that *s0* was taken by *a* is irrelevant.

Note also that parameter *d* goes into *s2* rather than sneaking into the unused upper bits of *v0*.

Since integer/pointer and floating point parameters are allocated independently, you can have multiple signatures that all use the same underlying calling convention.

```
void f(int i1, float f1);
void f(float f1, int i1);
```

In the next example, the structure `T` is a so-called *homogeneous floating-point aggregate*: It consists of a series of identical floating point types. It therefore is passable in floating point registers.

```
struct T { float x; float y; float a[2]; };
void f(T t1, float f1, T t2, float f2, int i);

    ; s0 = t1.x
    ; s1 = t1.y
    ; s2 = t1.a[0]
    ; s3 = t1.a[1]
    ; s4 = f
    ; t2 on the stack
    ; f2 on the stack
    ; w0 = i
```

The first parameter *t1* is passed in registers *s0* through *s3*. Next comes a float, which goes into *s4*. And then comes another `T`, but there are not enough registers remaining, so *t2* goes onto the stack. Note that *f2* also goes on the stack; it does not backfill into *s5*. On the other hand, we haven't run out of integer registers, so *i* can get passed in the low 32 bits of *x0*.

Varadic functions follow a different set of register assignment rules: Floating point registers are not used by variadic functions. 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