The AArch64 processor (aka arm64), part 1: Introduction

devblogs.microsoft.com/oldnewthing/20220726-00

July 26, 2022

Raymond Chen

The 64-bit version of the ARM architecture is formally known as AArch64. It is the 64-bit version of classic 32-bit ARM, which has been retroactively renamed AArch32.

Even though the architecture formally goes by the name AArch64, many people (including Windows) call it arm64. Even more confusing, the instruction set is called A64. (The 32-bit ARM instruction sets have also been retroactively renamed: Classic ARM is now called A32, and Thumb-2 is now called T32.)

AArch64 differs from AArch32 so much that I'm going to cover it fresh rather than treating it as an extension of AArch32. That said, I will nevertheless call out notable points of difference from AArch32.

No more Thumb mode

AArch64 is an extension of the classic ARM instruction set, not an extension of Thumb-2. So we're back to fixed-size 32-bit instructions (aligned on 4-byte boundaries). No more gymnastics with low registers and high registers, or using non-intuitive instructions to avoid a 32-bit encoding, or remembering to set the bottom bit on code addresses to avoid accidentally switching into classic mode.

A note for those familiar with the classic ARM instruction set: One thing that did not get carried forward was arbitrary predication. The answers to this StackOverflow question dig into the reasons why predication was removed. Short version: Predication is rarely used, it consumes a lot of opcode space, it doesn't interact well with out-of-order execution, and branch prediction is almost as good.

Data sizes

The architectural terms for data sizes are the same as AArch32.

Term	Size

byte	8 bits
halfword	16 bits
word	32 bits
doubleword	64 bits

The processor supports both big-endian and little-endian operation. Windows uses it exclusively in little-endian mode. AArch64 lost the Aarch32 `SETEND` instruction for switching endianness from user mode. Not that Windows supported it anyway.

Registers

Everything has doubled. The general-purpose registers are now 64 bits wide instead of 32. And the number of such registers has doubled from 16 to 32 okay just 31. The encoding that would correspond to register 31 has been reused for other purposes. So not quite doubled.

Register	Preserved?	Notes
x0	No	Parameter 1, return value
x1	No	Parameter 2
x2	No	Parameter 3
x3	No	Parameter 4
x4	No	Parameter 5
x5	No	Parameter 6
x6	No	Parameter 7
x7	No	Parameter 8
x8	No	
x9	No	
x10	No	
x11	No	
x12	No	
x13	No	
x14	No	

x15	No	
x16 (*xip0*)	Volatile	Intra-procedure call scratch register
x17 (*xip1*)	Volatile	Intra-procedure call scratch register
x18 (*xpr*)	read-only	TEB
x19	Yes	
x20	Yes	
x21	Yes	
x22	Yes	
x23	Yes	
x24	Yes	
x25	Yes	
x26	Yes	
x27	Yes	
x28	Yes	
x29 (*fp*)	Yes	frame pointer
x30 (*lr*)	No	link register
register “31” usually represents *sp* or *zr*, depending on instruction		

The link register is architectural; the rest are convention.

You can refer to the least significant 32 bits of each 64-bit register by changing the leading *x* to a *w*, so we have *w0* through *w30*. If an instruction targets a *w* register, the result is zero-extended to fill the *x* register.[1]

Particularly notable is that the stack pointer *sp* and program counter *pc* are no longer general-purpose registers, like they were in AArch32. The registers still exist, but they are treated as special registers rather than being encoded in the same way as the other general-purpose registers.

In AArch64, the *pc* special register reads as the address of the instruction being executed, rather than being four bytes ahead, as it was in AArch32. The extra +4 in AArch32 was an artifact of the internal pipelining of the original ARM and became a backward compatibility constraint even as the pipeline depth changed.

Windows requires that the stack remain 16-byte aligned, and it enables hardware enforcement of this requirement. The 32-bit subregister of *sp* is called *wsp*, although it is of no practical use. (The 64-bit register is still called *sp*, not *xsp*. Go figure.)

There is a 16-byte red zone below the stack pointer, but it's reserved for code analysis. Intrusive profilers inject assembly language fragments into compiled code to update profiling information, and they need some space to store two registers so they can free up some registers to do their profiling work.

The *xip0* and *xip1* registers are volatile because they are used to assist with branch instructions that try to branch to an address that is out of range. We'll see later that these registers are also used by function prologues and epilogues.

There is a new *xzr* pseudo-register (and its 32-bit alias *wzr*) which reads as zero, and writes are ignored. As I noted in the above table, if an instruction encodes a register number of 31, then a special behavior kicks in, typically by treating mythical register 31 as an alias for *sp* or *zr*. Generally speaking, when being used as a base address register, imaginary register 31 represents *sp*, but when used for arithmetic or as a destination register, it represents *zr*.[2]

In instruction descriptions, I will use these shorthands:

Shorthand	Meaning
`Xn`	Any *x#* register
`Xn/zr`	Any *x#* register or *xzr*
`Xn/sp`	Any *x#* register or *sp*
`Wn`	Any *w#* register
`Wn/zr`	Any *w#* register or *wzr*
`Wn/sp`	Any *w#* register or *wsp*
`Rn`	Any *x#* or *w#* register
`Rn/zr`	Any *x#* register, *w#* register, *xzr* or *wzr*

The floating point registers have been reorganized. They have doubled in size (to 128 bits) as well as in number, and the single-precision registers are no longer paired up.

Register	Preserved?	Notes
v0	No	Parameter 1, return value

v1	No	Parameter 2
v2	No	Parameter 3
v3	No	Parameter 4
v4	No	Parameter 5
v5	No	Parameter 6
v6	No	Parameter 7
v7	No	Parameter 8
v8 through *v15*	Low 64 bits only	Upper 64 bits are not preserved
v16 through *v31*	No	

Each floating point register can be viewed in multiple ways. The partial registers are stored in the least significant bits of the full register.

Name	Meaning	Notes
v#	SIMD vector	
q#	128-bit value	quad precision
d#	64-bit value	double precision
s#	32-bit value	single precision
h#	16-bit value	half precision
b#	8-bit value	

The flags register is formally known as the Application Program Status Register (APSR). The flags available to user mode are the same as in AArch32:

Mnemonic	Meaning	Notes
N	Negative	Set if the result is negative
Z	Zero	Set if the result is zero
C	Carry	Multiple purposes
V	Overflow	Signed overflow

Q	Saturation	Accumulated overflow
GE[n]	Greater than or equal to	4 flags (SIMD)

The overflow flag records whether the most recent operation resulted in signed overflow. The saturation flag is used by multimedia instructions to accumulate whether any overflow occurred since it was last cleared. The GE flags record the result of SIMD operations. By convention, flags are not preserved across calls.

There are a number of AArch64 features that you are extremely unlikely to see in Windows code, such as tagged pointers, tagged memory, and pointer authentication, so I won't cover them here. I also won't cover floating point instructions or SIMD instructions.

Next time, we'll look at some of the weird transformations that can be performed inside an instruction.

Additional references:

- Code in ARM Assembly: Registers explained. An analogous series looking at AArch64 from the Apple point of view rather than Windows.
- Writing ARM64 Code for Apple Platforms: The Apple ABI specification for AArch64.

[1] The Windows debugger isn't quite sure which name to use for these registers. The disassembler calls the registers *xip0*, *xip1*, and *xpr*, but the expression evaluator doesn't understand those names; you have to call them `@x16`, `@x17`, and `@x18`. On the other hand, the expression evaluator does understand `@fp` and `@lr` and refuses to acknowledge the existence of the names `@x29` and `@x30`. Furthermore, the expression evaluator doesn't understand any of the *w* aliases.

[2] AArch64's register 31 is similar to PowerPC's register 0, which changes meaning depending on the instruction. In PowerPC assembly, it was on you to keep track of which encodings treat register 0 as a value register, and which treat it as a zero register. At least AArch64 expresses the two cases differently: If an encoding uses pseudo-register 31 to mean *sp*, then you really must write *sp*. If you write *xzr*, you get an error.

PowerPC on the other hand would happily let you specify *r0* even if the instruction treats it as zero. Which was one of the jokes from the short-lived parody twitter account that mocked PowerPC.

> mscdfr - Means Something Completely Different For r0
>
> — PowerPC Instructions (@ppcinstructions) January 21, 2015

Raymond Chen

Follow