The AArch64 processor (aka arm64), part 19: Miscellaneous instructions

devblogs.microsoft.com/oldnewthing/20220822-00

August 22, 2022

Raymond Chen

There are far more instructions than I'm going to cover here in this series. I've skipped over the floating point instructions, the SIMD instructions, and specialty instructions that I haven't yet seen come out of the compiler. I'm also largely skipping over the instructions that are not part of the core instruction set but are available only in optional extensions.

Here are a few that are still interesting, even if I haven't seen the compiler generate them.

```
; count leading zeroes (high order bits)
clz     Rd, Rm          ; Rd = number of leading zeroes in Rm

; count leading sign bits (high order bits)
cls     Rd, Rm          ; Rd = number of leading sign bits in Rm

; reverse bits in register
rbit    Rd, Rm          ; Rd = Rm bitwise reversed

; reverse bytes in register
rev     Rd, Rm          ; Rd = Rm bytewise reversed

; reverse bytes in each halfword
rev16   Rd, Rm

; reverse bytes in each word
rev32   Rd, Rm

; reverse bytes in doubleword
; (pseudo-instruction, equivalent to rev with 64-bit register)
rev64   Rd, Rm
```

A few miscellaneous bit-fiddling instructions. The reversal instructions are primarily for changing data endianness. AArch64 lost the `REVSH` instruction from AArch32.

The next few instructions provide multiprocessing hints.

```
    ; yield to other threads
    yield

    ; wait for interrupt (privileged instruction)
    wfi
```

The `YIELD` instruction is a hint to multi-threading processors that the current thread should be de-prioritized in favor of other threads. You typically see this instruction dropped into spin loops, via the intrinsic `__yield()`.

The `WFI` instruction instructs the processor to go into a low-power state until an interrupt occurs. There are other instructions related to "events" which I won't bother going into.

The next few instructions are for communicating with the operating system:

```
        hlt     #imm16      ; halt
        svc     #imm16      ; system call
        brk     #imm16      ; software breakpoint
        udf     #imm16      ; undefined opcode
```

The instructions carry a 16-bit immediate that the operating system can choose to use for whatever purpose it desires.

The undefined opcode is a range of instructions from `0x00000000` through `0x0000ffff` that are architecturally set aside as permanently undefined instructions.[1]

Windows uses `BRK` for special operations.

```
        brk     #0xf000     ; breakpoint
        brk     #0xf001     ; assertion failure
        brk     #0xf002     ; debug service
        brk     #0xf003     ; fastfail
        brk     #0xf004     ; divide by zero
```

The divide-by-zero breakpoint is emitted by the compiler if it detects a zero denominator.

```
    cbnz    w0, @F          ; jump if denominator is nonzero
    brk     #0xf004         ; oops: manually raise div0 exception
@@: sdiv    w0, w1, w2      ; signed divide
```

And of course, we have this guy:

```
    nop
```

The `NOP` instruction does nothing but occupy space. Use it to pad code to meet alignment requirements, but do not use it for timing.

Now that we have the basic instruction set under our belt, we'll look at the calling convention next time.

[1] This means that zero encodes `udf #0`, which will trap on an invalid instruction. This is different from classic ARM, where zero encodes `andeq r0, r0, r0` which is functionally a nop, and Thumb-2, where zero is the `movs r0, r0` instruction, which is a mostly-nop except that it sets flags. Personally, I'm a big fan of having zero encode an invalid instruction. It helps post-mortem debugging a lot.

Raymond Chen

Follow