The AArch64 processor (aka arm64), part 17: Manipulating flags

devblogs.microsoft.com/oldnewthing/20220818-00

August 18, 2022

Raymond Chen

There is a pair of instructions for moving values into and out of the flags registers.

```
; move register from system register
; Rt = system_register
mrs     Rt/zr, system_register

; move system register from register
; system_register = Rt
msr     system_register, Rt/zr
```

There are a few system registers available. The one you want for accessing flags has the convenient name *nzcv*, which is named after the four bits it holds: Negative, Zero, Carry, and Overflow.

The only meaningful bits in *nzcv* are bits 28 through 31:

3 2									
9	8	7	6	5	4	3	2	1	0
N	Z	C	V						

The choice of bit positions is historical. That's where AArch32 put the bits in its version of the flags register, called *APSR*. And AArch32 put the bits there because, well, consider the story incorporated by reference.

The flag you typically want to manipulate is the carry, since it is an input to the `ADC` and `SBC` instructions. Forcing carry set or clear could be useful as a way to set initial conditions for multi-precision arithmetic.

One way to manipulate carry is to load the desired value into a register and write it to *nzcv*. But maybe we can find a cheaper way.

```
; clear carry by calculating 0+0
adds    wzr, wzr, #0

; equivalently...
cmn     wzr, #0

; set carry by calculating 0-0
subs    wzr, wzr, #0

; equivalently...
cmp     wzr, #0
```

Adding zero to zero does not incur unsigned overflow, so that clears carry. And subtracting zero from zero does not incur borrow, and since ARM uses true carry, this leaves carry set.

Instruction	N	Z	C	V
`cmn wzr, #0`	0	1	0	0
`cmp wzr, #0`	0	1	1	0
`cmp wzr, #1`	1	0	0	0

For fun, I also showed how to force the N flag set and force the Z flag clear. But I don't see how to force the V flag set in just one instruction.

If you want to toggle the carry bit, you can couple this with a conditional comparison.

```
; toggle carry (damages other flags)
ccmp wzr, #0, #0, cc
```

If carry is clear, then we perform a `cmp wzr, #0` which sets carry. If carry is set, then the *nzcv* value of zero is used, which clears carry (and everything else).

This trick tells us how we could set all the flags to any desired combination (including those not found in nature like "zero and negative"), but it'll take two instructions.

```
; clear carry to force next ccmp to fail
cmn     wzr, #0

; carry is clear, so the cmp never happens
; instead, flags are set to #n
ccmp    wzr, #0, #n, cs
```

First, we force carry clear with the magic `cmn wzr, #0` instruction. Then we perform a conditional comparison on carry set: Since carry is not set, the comparison is not performed, and instead, the flags are set according to `#n`.

When I introduced the condition codes, I noted that there is an encoding for *never*, but it doesn't work. Which is too bad, because a "never" encoding would have let us set flags to an arbitrary combination in a single instruction:

```
; This doesn't work
ccmp    wzr, #0, #n, nv
```

If this had worked, then the result would have been that the "never" test always fails, so we would always set the flags according to `#n`.

But it doesn't work, so oh well.

Next time, we'll look at some miscellaneous instructions that didn't fit easily into any of the categories so far.

Bonus chatter: An optional extension adds a few instructions for directly manipulating flags.

```
; carry flag invert (toggle carry flag)
; leaves other flags unchanged
cfinv

; rotate mask and insert flags
; set flags from 4 bits of a register
;
; if (mask & 8) N = Xn[lsb + 3];
; if (mask & 4) Z = Xn[lsb + 2];
; if (mask & 2) C = Xn[lsb + 1];
; if (mask & 1) V = Xn[lsb + 0];
;
rmif    Xn/zr, #lsb, #mask

; set flags based on 8-bit value
;
; N = Wn[7]
; Z = Wn[7:0] == 0
; C unchanged
; V = Wn[8] ^ Wn[7]
setf8   Wn/zr

; set flags based on 16-bit value
;
; N = Wn[15]
; Z = Wn[15:0] == 0
; C unchanged
; V = Wn[16] ^ Wn[15]
setf16  Wn/zr
```

It looks like the `SETF8` and `SETF16` instructions are for setting flags after performing arithmetic on sign-extended bytes or halfwords. You perform the arithmetic on the sign-extended value, and then use `SETF` on the result register to revise the flags to reflect the 8-bit or 16-bit result.

```
; load two sign-extended halfwords
ldrsh   r1, [r3]
ldrsh   r2, [r4]

; add them as words
add     r0, r1, r2

; set flags to match the halfword result
setf16  r0

bvs     signed_halfword_overflow
```

If you are going to be doing more signed sub-register math with the result, you probably want to perform an extra

```
; sign-extend the result so we can continue doing more math
sxth    r0, r0
```

Bonus chatter 2: Another special register is `PMCCNTR_EL0`, which is a 64-bit cycle counter.

Raymond Chen

Follow