The ARM processor (Thumb-2), part 14: Manipulating flags

devblogs.microsoft.com/oldnewthing/20210617-00

June 17, 2021

Raymond Chen

There are two instructions for accessing the flags register directly.

```
; move register from special register
mrs     Rd, apsr        ; Rd = APSR

; move special register from register
msr     apsr, Rd        ; APSR = Rd
```

These instructions are for accessing special registers, but the only special register available to user mode is `APSR`, so that's all you're going to see, if you even see this at all.

The format of the Application Program Status Register (APSR) is as follows:

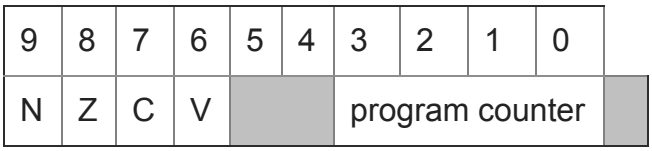
9	8	7	6	5	4	3	2	1	0	
N	Z	C	V	Q					GE[3:0]	

the N, Z, C, and V flags are updated by arithmetic operations. The GE flags are updated by SIMD operations. The Q flag is different: It is set when a saturating arithmetic operation overflows, and the only way to clear it is to issue an `MSR` instruction.

In user mode, the unlabeled bits of the APSR read as zero, and any attempts to modify them are ignored.

The odd placement of the four main numeric flags dates back to the first revision of the ARM processor.

The original ARM processor supported only 26-bit addresses, for a total address space of 64MB, and all instructions had to begin on a four-byte boundary. The unused bits of the *pc* register were repurposed to hold the flags!



The unlabeled bits are used only in kernel mode: In user mode, they read as zero and writes are ignored. The `Q` and `GE` flags had not been invented yet, so the only user-mode flags are N, Z, C, and V.

You can think of the flag bits as stowaways hiding inside the unused bits of the program counter register. If used as the first source parameter in a binary operation, all the extraneous non-program-counter bits were masked off, allowing you to perform *pc*-relative addressing and *pc*-based arithmetic.[1] In other contexts, however, the full 32-bit value of *pc* is used, flags and all.

When support expanded to a full 32-bit address space in ARM 3(?), those flag bits had to move to the APSR register, but to faciliate porting, their bit positions were preserved.

There are no dedicated instructions for manipulating specific flags. If you want to, say, set the carry flag and leave all other flags unchanged, you'll have to copy the ASPR to a general-purpose register, set the carry bit, and then set it back.

If you don't mind corrupting the other flags, then you can use some tricks to coerce a particular flag to a specific state.

```
; compare a number with itself
cmp     r0, r0      ; sets N = 0, Z = 1, C = 1, V = 0
```

Comparing a number sets flags according to the result of the subtraction, which produces zero. Therefore, the flags are set for nonnegative, zero, carry set (no underflow), and no overflow.

To clear carry, you can add zero:

```
adds    r0, r0, #0
```

Adding zero will never cause unsigned overflow, so this leaves carry clear.

Alternatively, if you don't want to create a false write dependency on *r0*, you could use

```
; add 0 and set flags, but discard result
cmn     r0, #0
```

This takes advantage of the lie hiding inside the `CMN` instruction that causes `CMN Rd, #0` to clear carry when it really should have set it.

If you want to force a nonnegative, zero result without affecting carry or overflow, you can use the otherwise-neglected `TEQ` instruction:

```
; test a number for equivalence with itself
teq     r0, r0      ; sets N = 0, Z = 1, C and V unchanged
```

To force a nonzero result, you can compare the stack pointer against an odd number, since Thumb-2 does not permit the stack pointer to be odd.

```
cmp     sp, #1      ; force nonzero result
```

You can't use *pc* for this trick because Thumb-2 does not allow the *pc* register to be used by a `CMP` instruction.

I couldn't think of a single-instruction way to force the negative or overflow bit to be set without modifying any integer registers. Maybe you can come up with something.[2]

Okay, so the second half of this article was mostly just code golf. Next time, we'll return to reality by looking at a few miscellaneous instructions.

[1] This explains the comment from Neil Rashbrook that you could use `TEQ` to copy the sign bit from a register into the *N* flag:

```
teq     pc, Rn      ; set flags according to (pc & 0x03FFFFFC) ^ Rn
```

Masking out the flag bits from the left-hand side (*pc*) means that the high bit is always clear. Exclusive-or with zero has no effect, so the tested value has the same high bit as *Rn*, which then becomes the *N* flag. This trick stopped working in ARM3, when the flags moved to a separate special register.

[2] I considered taking advantage of the fact that in Thumb-2 mode, the bottom bit of *pc* is always set, but the bit shifting and bit extraction instructions disallow *pc* as a source (or destination).

Raymond Chen