The ARM processor (Thumb-2), part 6: The lie hiding inside the CMN instruction

devblogs.microsoft.com/oldnewthing/20210607-00

June 7, 2021

Raymond Chen

Last time, we learned that the `CMN` instruction stands for *compare negative*, and it compares its first argument with the negative of the second argument:

```
; compare negative (compare Rn with -op2)
cmn     Rn, op2             ; Set flags for Rn + op2
```

We noted that the `N` in the name is misleading, because it stands for *negative*, even though in the seemingly-analogous `MVN` instruction, the `N` stands for *not*.

But that's not the most misleading part of the `CMN` instruction.

The big lie about the *compare negative* instruction is that *it doesn't even compare the negative*.

You had one job!

The *compare negative* instruction is defined in terms of addition, rather than subtraction of the negative. Mathematically, the operations are identical: Subtracting a negative is the same as adding the positive.

But computers aren't operating on mathematical integers.

Let's look more closely at the difference between subtraction of the negative and addition of the positive. Recall that subtraction in a true-carry system is rewritten as addition, using the identity `-x = ~x + 1`. Therefore, `a - b` becomes `a + ~b + 1`. And the carry for this operation is the combined carry from *both* additions.

On the other hand, the `CMN` instruction is implemented as a straight addition, not a subtraction of the negative.

Expression	Evaluated as

`a - (-b)`	let `c = ~b + 1` result `= a + ~c + 1`
`a + b`	result `= a + b`

One difference between the two is that in the “subtract the negative” version, the carry that occurs in the calculation of `~c + 1` contributes to the final carry, whereas that extra carry disappears in the “add the positive” version.

When would `~c + 1` generate a carry?

Answer: When `c` is zero, because we have `~c + 1 = ~0 + 1 = 0xFFFFFFFF + 1`, and that sum generates a carry. And `c` is zero when `b` is zero.

On the other hand, if `b` is zero, then `a + b` never generates a carry because you’re just adding zero, which does nothing.

Conclusion: If the second parameter to `CMN` is zero, then `CMN` results in no carry, even though “subtracting the negative of zero” would have produced a carry.

In other words, these two sequences do not generate the same flags:

```
    ; compare r0 against negative r1 using single instruction
    cmn     r0, r1          ; set flags for (r0 + r1)

    ; compare r0 against negative r1 using two instructions
    rsb     r12, r1, #0     ; r12 = negative r1
    cmp     r0, r12         ; set flags for (r0 - r12)
```

In the case where *r1* is zero, the first version clears carry, but the second version sets carry. This means that if you follow the `CMN` with a conditional branch that consumes carry, you will get the wrong answer if the second parameter happens to be zero.

We haven’t learned about ARM conditionals yet, but when we do, you’ll discover that it is the unsigned relative comparison conditionals[1] that are based on the carry flag. Therefore, don’t use the `CMN` instruction to make unsigned relative comparisons against values which might be zero, because the carry flag may be set incorrectly.

Fortunately, this problem is relatively easy to avoid:

First rule: Don’t use

```
    cmn     Rd, #0
```

Don’t hard-code zero as the second parameter because the carry won’t be set properly.[2] Just write it as the positive comparison:

```
    cmp     Rd, #0
```

This is even easier to write, and it doesn't have the carry flag problem.

Second rule: If you use a register or shifted register as the second parameter to `CMN`, don't follow it with a condition that relies on the carry flag. In practice, this means that you should use signed conditions to test the result rather than unsigned conditions. (We haven't learned about conditions yet.)

Fortunately, this second rule is not that much of a problem, because the fact that you are "comparing against the negative" strongly implies that you are interpreting the comparison as between two signed integers, so you are unlikely to follow up with an unsigned relative comparison conditional.

Okay, so that takes care of carry. The other troublesome bit is the overflow bit, which is just the carry from bit 30 into 31.

What are the cases in which a carry from bit 30 to bit 31 would be lost? We already identified zero as one of those cases. The other case is where the value is `0x80000000`.

Ah, the curse of the most negative integer.

Ignoring the overflow from bit 30 to bit 31 means that the negative of `0x80000000` is treated as the positive number `+0x80000000`. Since every two's complement 32-bit integer is less than `+0x80000000` (viewed as a large positive number), the result of the comparison is pretty much foregone. It will always say "less than".

In a way, though, the curse of the most negative integer isn't so much of a problem here. After all, you did ask to compare against the negative of the most negative integer, and that's a positive number that's so positive it can't even be represented!

Wow, what a complicated, messed-up instruction `CMN` turned out to be.

Next time, we'll return to our exploration of the ARM Thumb-2 instruction set by looking at bitwise operations.

[1] The unsigned relative comparison conditionals are "unsigned less than", "unsigned less than or equal", "unsigned greater than", and "unsigned greater than or equal".

[2] Note that everything is fine if the second argument is not zero.

```
    ; compare against 0xFFFFFFFE.
    ; this works properly even for unsigned comparison.
    cmn     r1, #2
```

The problematic case for unsigned comparisons occurs only when the second argument is zero.

Raymond Chen

Follow