The ARM processor (Thumb-2), part 5: Arithmetic

devblogs.microsoft.com/oldnewthing/20210604-00

June 4, 2021

Raymond Chen

The general format of three-register instructions in Thumb-2 goes like this:[1]

```
op      Rd, Rn, #imm12      ; Rd = Rn op decode(imm12)
op      Rd, Rn, Rm          ; Rd = Rn op Rm
op      Rd, Rn, Rm, shift   ; Rd = Rn op (Rm with shift applied)
                            ; shift can be LSL, LSR, ASR, ROR
```

The `#imm12` is a constant in a form we discussed last time.

For notational convenience, let's call this

```
op      Rd, Rn, op2         ; op2 can be #imm12, Rm, or Rm with a shift
```

Sometimes you'll see a two-register version, which is shorthand for (and often a more compact encoding than) the three-register version:

```
op      Rd, Rn              ; shorthand for op Rd, Rd, Rn
```

Like the PowerPC, the ARM uses true carry. This means that for subtraction, the carry is clear when a borrow occurs, and subtract with carry subtracts an additional unit if inbound carry is clear.

With that said, here are the basic arithmetic operations:

```
; add
add     Rd, Rn, op2         ; Rd = Rn + op2

; add with carry
adc     Rd, Rn, op2         ; Rd = Rn + op2 + carry

; subtract
sub     Rd, Rn, op2         ; Rd = Rn - op2

; subtract with carry
sbc     Rd, Rn, op2         ; Rd = Rn - op2 - !carry

; reverse subtract
rsb     Rd, Rn, op2         ; Rd = op2 - Rn

; reverse subtract with carry
rsc     Rd, Rn, op2         ; Rd = op2 - Rn - !carry

; copy register from constant, register, or generalized op2
mov     Rd, #imm8           ; Rd = imm8 (0 to 255)
mov     Rd, Rm              ; Rd = Rm
mov     Rd, op2             ; Rd = op2

; copy register from bitwise NOT of register or generalized op2
mvn     Rd, Rm              ; Rd = ~Rm
mvn     Rd, op2             ; rd = ~op2

; all support the S suffix
```

I noted earlier that in traditional RISC, there is no need for an architectural `MOV` instruction because you can treat it as a pseudo-instruction formed by adding zero to a register. Thumb-2 does include it as a special instruction because it has a 16-bit encoding in the case where you are loading a small positive constant, or if you are copying to a low register (even if the source register is high). There's also a more traditional `op2` format that takes decoded 12-bit immediates or shifted registers.

The most valuable part of reverse subtraction is that you can use it to subtract from a constant. In particular, you can negate a register by subtracting it from zero.

There are also discarding versions of the subtraction instructions, where the sole purpose is setting flags.

```
; compare (compare Rn with op2)
cmp     Rn, op2             ; Set flags for Rn - op2

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

The ARM processor designers are pulling a fast one here. In the `MVN` instruction, the `N` stands for *not*, meaning that it moved the bitwise negation of the `op2`. But in `CMN`, the `N` stands for *negative*, meaning that it compares the arithmetic negative of the `op2`.

There's an even more devious trap hiding in the `CMN` instruction, which I will discuss next time.

Multiplication has a few variations. These are the 32 × 32 → 32 multiplies:

```
; multiply
mul     Rd, Rn, Rm          ; Rd = Rn * Rm
muls    Rd, Rn, Rm          ; Rd = Rn * Rm, set partial flags

; multiply accumulate
mla     Rd, Rm, Rs, Rn      ; Rd = (Rm * Rs) + Rn

; multiply subtract
mls     Rd, Rm, Rs, Rn      ; Rd = Rn - (Rm * Rs)
```

The only multiply or divide instruction that has the option to set flags is `MULS`. It updates the negative (N) and zero (Z) flags to match the result, but the carry (C) and overflow (V) flags are unmodified.

And here are the 32 × 32 → 64 multiplies:

```
; unsigned multiply long
umull   Rdlo, Rdhi, Rm, Rs  ; Rdhi:Rdlo = Rm * Rs (unsigned)

; signed multiply long
smull   Rdlo, Rdhi, Rm, Rs  ; Rdhi:Rdlo = Rm * Rs (signed)

; unsigned multiply accumulate long
umlal   Rdlo, Rdhi, Rm, Rs  ; Rdhi:Rdlo = Rdhi:Rdlo + Rm * Rs (unsigned)

; signed multiply accumulate long
smlal   Rdlo, Rdhi, Rm, Rs  ; Rdhi:Rdlo = Rdhi:Rdlo + Rm * Rs (signed)

; unsigned multiply accumulate accumulate long
umaal   Rdlo, Rdhi, Rm, Rs  ; Rdhi:Rdlo = Rdhi + Rdlo + Rm * Rs (unsigned)
```

The "unsigned multiply accumulate accumulate long" instruction is a bit of an oddball. Its funny name reflects the fact that the registers of the output register pair are treated as separate integer inputs.

Of the multiply instructions, I've seen the compiler use `MUL`, `MLA`, `UMULL` and `SMULL`. I have yet to see it use `UMLAL`, `SMLAL`, or `UMAAL`.

There are also division instructions, but they are architecturally optional and raise an "invalid instruction" on processors that don't support them.

```
; unsigned divide
udiv    Rd, Rn, Rm          ; Rd = Rn / Rm (unsigned)

; signed divide
sdiv    Rd, Rn, Rm          ; Rd = Rn / Rm (signed)
```

The division instructions perform integer unsigned or signed division, with the result rounded toward zero. In the special case of signed division of `0x80000000 ÷ 0xFFFFFFFF`, the processor produces a result of `0x80000000` without trapping. By default, division by zero does not trap; it just returns zero. However, some revisions allow the operating system to enable trapping on division by zero. Windows enables trapping when the processor supports it.[2]

If hardware support for division is not present, the instructions trap into the kernel, where the operation is emulated. Operating system code generally does not assume hardware division support, and division will call out to a helper function to perform the division.

I'm skipping over the SIMD and multimedia instructions, like saturating arithmetic and parallel arithmetic. I have yet to see them in compiler-generated code.

Next time, we'll look at the lie hiding inside the `CMN` instruction.

Bonus chatter: Commenter Petteri Aimonen points out that even though the division operation does not produce the remainder, you can recover the remainder with just one additional instruction, thanks to the "multiply and subtract" instruction:

```
sdiv    Rq, Rn, Rm          ; Rq = Rn / Rm (signed)
mls     Rr, Rq, Rm, Rn      ; Rr = Rn - (Rq * Rm) = Rn % Rm
```

In practice, the MSVC, gcc and clang compilers default to assuming that `sdiv` is an emulated instruction and performing the division manually rather than risking a trap. The emulated version produces the remainder for free as a by-product. If you tell them to assume armv7ve, then they will enable the native division instruction. The gcc and clang compilers will use `mls` to calculate the remainder. MSVC breaks it into separate `mul` and `subs` instructions.

[1] Classic ARM also supports shifting by an amount provided by a fourth register, leading to instructions like

```
ADD     Rd, Rn, Rm, LSL Rs  ; Rd = Rn + (Rm << Rs)
```

[2] There is no dedicated "divide by zero" trap. Instead, if division by zero is attempted, the processor raises an "invalid instruction" trap. The trap handler is expected to parse the faulting instruction, identify it as a valid division instruction, and then realize that the divisor is zero.

Raymond Chen

Follow