The AArch64 processor (aka arm64), part 5: Multiplication and division

devblogs.microsoft.com/oldnewthing/20220801-00

August 1, 2022

Raymond Chen

There are a lot of ways of multiplying two values. The most basic way is to multiply two registers of the same size, producing a result of the same size.

```
; multiply and add
; Rd = Ra + (Rn × Rm)
madd    Rd/zr, Rn/zr, Rm/zr, Ra/zr

; multiply and subtract
; Rd = Ra - (Rn × Rm)
msub    Rd/zr, Rn/zr, Rm/zr, Ra/zr
```

The product is then added to or subtracted from a third register.

You get some pseudo-instructions if you hard-code the third input operand to zero.

```
; multiply
mul     a, b, c                         ; madd a, b, c, zr

; multiply and negate
mneg    a, b, c                         ; msub a, b, c, zr
```

The next fancier way of multiplying two registers is to multiply two 32-bit registers and get a 64-bit result.

```
; unsigned multiply and add long
; Xd = Xa + (Wn × Wm), unsigned multiply
umaddl  Xd/zr, Wn/zr, Wm/zr, Xa/zr

; unsigned multiply and subtract long
; Xd = Xa - (Wn × Wm), unsigned multiply
umsubl  Xd/zr, Wn/zr, Wm/zr, Xa/zr

; signed multiply and add long
; Xd = Xa + (Wn × Wm), signed multiply
smaddl  Xd/zr, Wn/zr, Wm/zr, Xa/zr

; signed multiply and subtract long
; Xd = Xa - (Wn × Wm), signed multiply
smsubl  Xd/zr, Wn/zr, Wm/zr, Xa/zr
```

Again, the result of the multiplication is added to or subtracted from an accumulator. The naming of this opcode is a little confusing, because the word *long* in the opcode talks about the multiplication, not the addition or subtraction. The multiplication is 32 × 32 → 64, and the result is then accumulated as a 64-bit value.

You can probably guess what the pseudo-instructions are. Just hard-code the zero register as the accumulator.

```
; unsigned multiply long
umull   a, b, c                    ; umaddl a, b, c, zr

; unsigned multiply and negate long
umnegl  a, b, c                    ; umsubl a, b, c, zr

; signed multiply long
smull   a, b, c                    ; smaddl a, b, c, zr

; signed multiply and negate long
smnegl  a, b, c                    ; smsubl a, b, c, zr
```

The last multiplication instruction gives you the missing piece of the 64 × 64 → 128 multiply.

```
; unsigned multiply high
; Xd = (Xn × Xm) >> 64, unsigned multiply
umulh   Xd/zr, Xn/zr, Xm/zr

; signed multiply high
; Xd = (Xn × Xm) >> 64, signed multiply
smulh   Xd/zr, Xn/zr, Xm/zr
```

These give you the upper 64 bits of a 64 × 64 → 128 multiply. If you want the full 128 bits, you combine it with the corresponding 64 × 64 → 64 multiply to get the lower 64 bits.

```
; unsigned 64 × 64 → 128
; r1:r0 = r2 × r3
mul     r0, r2, r3
umulh   r1, r2, r3

; signed 64 × 64 → 128
; r1:r0 = r2 × r3
mul     r0, r2, r3
smulh   r1, r2, r3
```

Don't be fooled by the lack of symmetry: Even though there is a `UMULL` instruction, it is not the counterpart to `UMULH`, and `SMULL` instruction is not the counterpart to `SMULH`!

Whereas there are a large variety of ways to multiple two registers, there are only two ways to divide them.

```
; unsigned divide
; Rd = Rn ÷ Rm, unsigned divide, round toward zero
udiv    Rd/zr, Rn/zr, Rm/zr

; signed divide
; Rd = Rn ÷ Rm, signed divide, round toward zero
sdiv    Rd/zr, Rn/zr, Rm/zr
```

If you try to divide by zero, there is no exception. The result is just zero. If you want to trap division by zero, you'll have to test for a zero denominator explicitly.

There is also no exception for dividing the most negative integer by –1. You just get the most negative integer back.

None of the multiplication or division operations set flags.

There is no instruction for calculating the remainder. You can do that manually by calculating $r = n - (n \div d) \times d$. This can be done by following up the division with an `msub`:

```
; unsigned remainder after division
udiv    Rq, Rn, Rm          ; Rq = Rn ÷ Rm
msub    Rr, Rq, Rm, Rn      ; Rr = Rn - Rq × Rm
                            ;    = Rn - (Rn ÷ Rm) × Rm

; signed remainder after division
sdiv    Rq, Rn, Rm          ; Rq = Rn ÷ Rm
msub    Rr, Rq, Rm, Rn      ; Rr = Rn - Rq × Rm
                            ;    = Rn - (Rn ÷ Rm) × Rm
```

Next time, we'll look at the logical operations and their extremely weird immediates.

Raymond Chen

Follow