The AArch64 processor (aka arm64), part 4: Addition and subtraction

devblogs.microsoft.com/oldnewthing/20220729-00

July 29, 2022

Raymond Chen

Most of the binary operation instructions are of the form

```
    op      x, y, z         x = y op z
```

They take two source operands, combine them according to some operation, and put the result in the destination register.

Similarly, most of the unary operation instructions look like

```
    op      x, y            ; x = op y
```

The destination is typically a numbered register or *sp*, and can be a 64-bit register or a 32-bit subregister. If you use a 32-bit subregister, then the result is zero-extended to a 64-bit value.

Okay, let's start with addition:

```
    add     Rd/sp, Rn/sp, #imm12
    add     Rd/sp, Rn/sp, #imm12, LSL #12

    add     Rd/zr, Rn/zr, Rm/zr, LSL #n
    add     Rd/zr, Rn/zr, Rm/zr, LSR #n
    add     Rd/zr, Rn/zr, Rm/zr, ASR #n

    add     Rd/sp, Rn/sp, Rm/zr, UXTB #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, UXTH #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, UXTW #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, UXTX #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, SXTB #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, SXTH #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, SXTW #n    ; 0 ≤ n ≤ 4
    add     Rd/sp, Rn/sp, Rm/zr, SXTX #n    ; 0 ≤ n ≤ 4
```

To ask for flags to be set based on the result, apply an `S` suffix to the opcode, producing `ADDS`.

Note that some of these encodings permit the operand to be `sp`, but others allow `zr`.

The first two versions add an immediate. It is either a 12-bit unsigned immediate ($0 \le n \le 4095$) or a 12-bit unsigned immediate shifted left by 12. This means that you can express constants of the form `0x00000XXX` and `0x00XXX000`. The disassembler does the `LSL #12` for you, so you won't actually see the `#imm12, LSL #12` version disassembled as such. Instead, you'll see the shifted constant:

```
    add     x0, x1, #0x123000   ; encoded as #0x123, LSL #12
```

The next block of variants adds a shifted register. You are allowed to shift doublewords by up to 63 positions and words up to 31 positions. You don't need any larger shifts, because unsigned shifting by an amount greater than or equal to the operand bit size just gives you zero, so you should just have used *zr*. And signed shifting right by an amount greater than or equal to the operand size is the same as shifting right by one less than the operand bit size.

The last block lets you use the extended registers. You can use all of the extended forms, and the shift amount can be up to four positions. These extended registers with shifts are convenient for calculating array offsets:

```
    ; x0 = x1 + (int32_t)x2 * 16
    add     x0, x1, x2, SXTW #4
```

In this case, *x1* is the base of an array where each element is of size 16, and *x2* is a 32-bit signed array index, and we calculate the address of the element into *x0*.

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.

The subtraction instruction has the same available variants as the addition instructions.

```
    ; calculate x = y - z
    sub     x, y, z     ; same options as add

    ; calculate x = y - z, set flags
    subs    x, y, z     ; same options as adds
```

Adding and subtracting with carry have only one encoding option.

```
    ; Rd = Rn + Rm + carry
    adc     Rd/zr, Rn/zr, Rm/zr

    ; Rd = Rn + Rm + carry, set flags
    adcs    Rd/zr, Rn/zr, Rm/zr

    ; Rd = Rn - Rm - !carry
    sbc     Rd/zr, Rn/zr, Rm/zr

    ; Rd = Rn - Rm - !carry, set flags
    sbcs    Rd/zr, Rn/zr, Rm/zr
```

From the addition and subtraction instructions, we can construct these pseudo-instructions, taking advantage of literal zeros and the hard-coded zero register: Reads from the zero register produce zero, and writes to the zero register are discarded.

```
; move register to/from sp
mov     sp, Rn               ; add sp, Rn, #0
mov     Rn, sp               ; add Rn, sp, #0

; move constant to register
mov     Rn, #imm12           ; add Rn, zr, #imm12
mov     Rn, #imm12, LSL #12 ; add Rn, zr, #imm12, LSL #12
```

Adding zero gives you the ability to move between *sp* and the general-purpose registers. And adding an immediate to the zero register loads a constant. We'll see later that other register-to-register moves are encoded with a different pseudo-instruction, and there are plenty of options for loading constants beyond just this one.

The use of true carry permits the following group of pseudo-instructions for adding or subtracting negative numbers:

```
add     a, b, #-n            ; sub  a, b, #n
adds    a, b, #-n            ; subs a, b, #n

sub     a, b, #-n            ; add  a, b, #n
subs    a, b, #-n            ; adds a, b, #n
```

The immediate operand to the `ADD` and `SUB` instruction families is treated as unsigned, but you can switch to the opposite instruction to get negative values (provided $n \neq 0$). Note that this works due to ARM's use of true carry. (If ARM had used borrow, then this conversion would set the carry bit incorrectly.)

```
cmp     x, y                 ; subs zr, x, y
cmn     x, y                 ; adds zr, x, y
```

The *compare* and *compare negative* instructions are just subtraction and addition that set flags and throw away the result. Beware of the lie hiding inside the CMN instruction.

```
; negate (possibly setting flags)
neg     x, y, shift          ; sub  x, zr, y, shift
negs    x, y, shift          ; subs x, zr, y, shift

; negate with carry (possibly setting flags)
ngc     x, y, shift          ; sbc  x, zr, y, shift
ngcs    x, y, shift          ; sbcs x, zr, y, shift
```

Subtracting from zero gives you the ability to negate a value. Note that these pseudo-instructions are available only with shifted registers because the corresponding subtraction instructions support *zr* as the first input only when the second input is a shifted register. (Of course, you can shift by *#0* if you didn't really want to shift the second input.)

That turned out to be a lot to say about addition and subtraction. Next time, we'll look at the fancier arithmetic operations: Multiplication and division.

Raymond Chen

Follow