The ARM processor (Thumb-2), part 4: Single-instruction constants

devblogs.microsoft.com/oldnewthing/20210603-00

June 3, 2021

Raymond Chen

I noted last time that ARM is very proud of its barrel shifter. We saw it being used in the effective address calculator. Another place it makes itself known is in the calculation of constants in a single-instruction.

First, the easy case: An unsigned 8-bit immediate, which gives you constants 0 through 255.

```
movs    Rd, #imm8       ; Rd = imm8 and set some flags
```

Only the sign flag (N) and zero flag (Z) are updated to match the value. The carry (C) and overflow (V) flags are unaffected. This 16-bit encoding is available only for low registers.

ARM has dedicated instructions for loading constants. In classic RISC architectures, loading constants is typically done by performing arithmetic against a hard-coded zero register:

```
lda     Rd, nnnn(zero)  ; Alpha AXP
addi    rd, zero, nnnn  ; MIPS
addi    rd, 0, nnnn     ; PowerPC
```

ARM doesn't have a dedicated zero register, and it doesn't have enough encoding space for a 16-bit immediate, so it has to come up with something else.

It does it by showing off its barrel shifter.

You can take your 8-bit unsigned immediate and shift it left by up to 24 positions, thereby allowing you to create any 32-bit constant where the span from the lowest to highest set bit is at most 8 positions.

There are also a few special transformations:

Pattern	Notes
`0x000000AB`	Copy to positions 7:0 (nop).

`0x00AB00AB`	Copy to positions 23:16 and 7:0.
`0xAB00AB00`	Copy to positions 31:24 and 15:8.
`0xABABABAB`	Copy to all bytes.

These special transforms are handy for setting up a register to fill memory with a repeating pattern. For example,

```
mov     r0, #0x20202020     ; ASCII spaces
mov     r0, #0x00200020     ; UTF-16 spaces
mov     r0, #0xfefefefe     ; debug fill value
```

The naïve way of encoding these constants would be to have an 8-bit immediate and a 5-bit shift amount (to encode shifts 0 through 24), and using shifts above 24 to encode the special transformations, for a total encoding space of 8 + 5 = 13 bits. But Thumb-2 manages to encode all of these constants in just 12 bits.

The trick is to realize that the 8 + 5 encoding has a lot of redundancy. The constant 4096, for example, could be encoded eight different ways. It could be `1 << 12`, or `2 << 11`, up to `128 << 5`. The Thumb-2 encoding exploits this redundancy by requiring that the 8-bit value being shifted have a 1 in bit 7.[1] This forces a unique representation for all of the shift scenarios (in our case, `128 << 5`), and allows bit 7 of the constant to be used to help encode the shift amount.

Related reading: How did real-mode Windows patch up return addresses to discarded code segments? Another example of squeezing ten pounds of flour into a five-pound bag.

```
mov     Rd, #imm12      ; Rd = decode(imm12)
movs    Rd, #imm12      ; Rd = decode(imm12), set some flags
```

If you ask for flags to be updated, then the sign flag (N) and zero flag (Z) are updated to match the generated constant. The overflow (V) flag is unchanged, and the carry (C) flag is updated in a complicated way you probably don't care about.[2]

But wait, we're not done with generating constants:

```
mvn     Rd, #imm12      ; Rd = ~decode(imm12)
mvns    Rd, #imm12      ; Rd = ~decode(imm12), set some flags
```

In addition to all of the special constants that can be generated with `MOV`, you can use the `MVN` instruction to generate the bitwise NOT of them all.[3]

The `MVN` is commonly used to generate small negative numbers:

```
mvn     Rd, #0          ; Rd = -1
mvn     Rd, #1          ; Rd = -2
```

But wait, we're still not done yet!

```
mov     Rd, #imm16      ; Rd = imm16
```

There is also a special encoding that loads a 16-bit unsigned value.

As we saw last time, if you're writing assembly by hand, you can just write `LDR Rd, =#nnn` and the assembler will figure out which `MOV`, `MVN`, or (worst case) `LDR` instruction will get you the value you want. It will disassemble as `MOV`, `MVN`, or `LDR` based on what the assembler ultimately chose.

Finally, there's another constant-generating function for replacing the upper 16 bits of a register:

```
; move top
movt    Rd, #imm16      ; Rd[31:16] = imm16
                        ; Rd[15: 0] unchanged
```

The upper 16 bits of the destination register are replaced by the 16-bit immediate, and the lower 16 bits are left unchanged. This instruction is usually paired with the `#imm16` version of the `MOV` instruction:

```
mov     Rd, #efgh
movt    Rd, #abcd       ; Rd = abcdefgh
```

Of all these ways of generating constants, the `#imm12` constants can also be used as immediate arguments to arithmetic operations. We'll start looking at those arithmetic operations next.

[1] This means that you cannot use the shift-encoding format for constants less than 128. But that's okay, because those constants can use the "nop" transformation.

[2] It's an artifact of the way the constant is generated internally: If you request a special transformation (or no transformation), then the carry flag is unchanged. If you request a shift, it is internally treated as a *rotate right* of the unshifted value, and the carry flag consequently matches the high bit of the result.

[3] If you request flags to be updated, then the sign and zero flags reflect the result *after* bitwise negation, but the carry flag reflects the result *before* bitwise negation.

Raymond Chen

Follow