The ARM processor (Thumb-2), part 8: Bit shifting and bitfield access

devblogs.microsoft.com/oldnewthing/20210609-00

June 9, 2021

Raymond Chen

The ARM processor shows off its barrel shifter once again in its collection of bit shifting instructions.

```
; logical shift right
lsr     Rd, Rn, #imm5       ; Rd = Rn >> imm5        (unsigned)
lsr     Rd, Rn, Rm          ; Rd = Rn >> (Rm & 0xFF) (unsigned)

; arithmetic shift right
asr     Rd, Rn, #imm5       ; Rd = Rn >> imm5        (signed)
asr     Rd, Rn, Rm          ; Rd = Rn >> (Rm & 0xFF) (signed)

; logical shift left
lsl     Rd, Rn, #imm5       ; Rd = Rn << imm5
lsl     Rd, Rn, Rm          ; Rd = Rn << (Rm & 0xFF)

; rotate right
ror     Rd, Rn, #imm5       ; Rd = rotate_right(Rn, imm5)
ror     Rd, Rn, Rm          ; Rd = rotate_right(Rn, Rm & 0xFF)

; rotate right extended
rrx     Rd, Rn              ; temp = Rn
                            ; Rd = (carry << 31) | (temp >> 1)
                            ; carry = temp & 1

; all support the S suffix
```

For register-counted shifts, only the bottom byte of the shift amount is used. The “rotate right extended” instruction performs a 33-bit rotation, where the carry bit is the extra bit.

If flags are updated, then the negative (N) and zero (Z) flags reflect the resulting value. The carry (C) flag contains the last bit shifted out. and the overflow (V) flag is unchanged. If the shift amount is zero, then carry is unchanged.

There is no `RLX` instruction for rotating left through carry, but that’s okay, because you can emulate it:

```
    adcs    Rd, Rn, Rn          ; Rd = Rn + Rn + carry, set carry on overflow
```

Adding a number to itself is the same as shifting left one position. Adding with carry puts the former carry bit into bit 0 of the result. And setting flags on carry-out means that the previous bit 31 becomes the new carry bit. Voilà: Rotate left through carry.

Note that for the shift instructions, the shift amount cannot itself be a shifted register. The barrel shifter is already being used by the primary opcode; it can't be used to generate the shift amount, too.

There are also some instructions specifically for manipulating bitfields.

```
    ; bitfield clear: zero out #w bits starting at #lsb
    bfc     Rd, #lsb, #w        ; Rd[lsb+w-1:lsb] = 0

    ; bitfield insert: replace #w bits in starting at #lsb
    ; with least significant bits of source
    bfi     Rd, Rn, #lsb, #w    ; Rd[lsb+w-1:lsb] = Rn[w-1:0]

    ; unsigned bitfield extract
    ubfx    Rd, Rn, #lsb, #w    ; Rd = Rn[lsb+w-1:lsb], zero-extended

    ; signed bitfield extract
    sbfx    Rd, Rn, #lsb, #w    ; Rd = Rn[lsb+w-1:lsb], sign-extended
```

Suppose you have a C structure like this:

```
struct S
{
    int x:10;
    int y:12;
    unsigned int z:10;
};
```

which might correspond to

9	8	7	6	5	4	3	2	1	0		
z										y	x

Suppose the bitfield is held in register *r0* and the variable *v* is in the register *r1*. The bitfield instructions would correspond to these C statements:

```
    bfc     r0, #10, #12        ; s.y = 0

    bfi     r0, r1, #10, #12    ; s.y = v

    ubfx    r1, r0, #22, #10    ; v = s.z
    sbfx    r1, r0, #10, #12    ; v = s.y
```

The “bitfield clear” instruction sets a range of bits to zero. The “bitfield insert” instruction copies the specific number of least significant bits of the source to a position in the destination. The bitfield extraction instructions copy the specific bits from the source to the least significant bits of the destination, and either zero-extends or sign-extends the result.

The bitfield clear instruction can also be used for things other than bitfields, For example, you can write

```
bfc     r0, r0, #14, #18    ; r0 = r0 & 0x0003FFFF
```

You would be tempted to write something like

```
and     r0, r0, #0x0003FFFF ; not a valid instruction
```

but if you try, the assembler will get mad at you because the constant `0x0003FFFF` cannot be encoded. There are too many 1-bits for it to be encoded as a shifted 8-bit value, and there are too many 0-bits for it to be encoded as the inverse of a shifted 8-bit value.

The signed bit field extraction instruction is useful for sign-extending a sub-word value in a single instruction:

```
sbfx    r0, r0, #0, #12     ; sign extend a 12-bit value

; alternative version would have been
lsl     r0, r0, #20         ; r0 = r0 << 20
asr     r0, r0, #20         ; r0 = r0 >> 20 (signed)
```

The bitfield instructions use a 32-bit encoding. While you could use them to sign-extend or zero-extend a byte or halfword, there are dedicated 16-bit instructions for those operations. We’ll look at those next time.

Raymond Chen

Follow