The AArch64 processor (aka arm64), part 8: Bit shifting and rotation

devblogs.microsoft.com/oldnewthing/20220804-00

August 4, 2022

Raymond Chen

Bit shifting and rotation instructions on AArch64 fall into two general categories: Hard-coded shift amounts and variable shifts.

The hard-coded shifts are done by repurposing the versatile bitfield manipulation instructions.

```
; logical shift left by fixed amount
; ubfiz Rd, Rn, #(size-shift), #shift
lsl     Rd/zr, Rn/zr, #shift

; logical shift right by fixed amount
; ubfx  Rd, Rn, #(size-shift), #shift
lsr     Rd/zr, Rn/zr, #shift

; arithmetic shift right by fixed amount
; sbfx  Rd, Rn, #(size-shift), #shift
asr     Rd/zr, Rn/zr, #shift
```

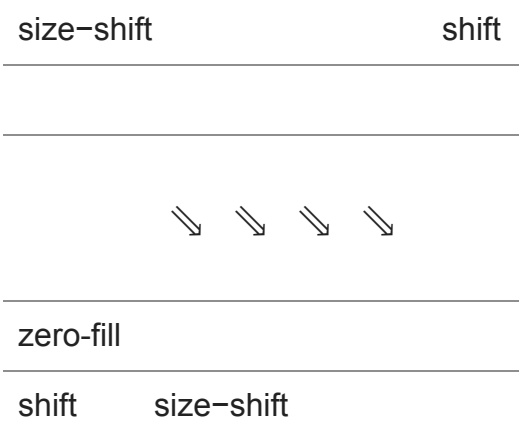
Left shifting is done by doing a bit insertion of the surviving bits into the upper bits of the destination. It's the special case where the number of bits is exactly equal to the register size minus the shift amount.

shift	size−shift

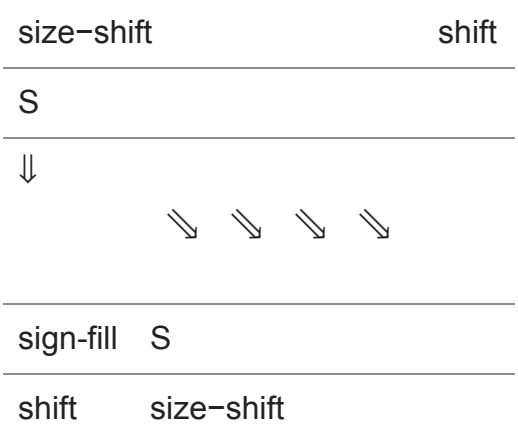
↙ ↙ ↙ ↙

	zero-fill
size−shift	shift

Right shifting is the same thing, but using the unsigned bitfield extract instruction to go in the opposite direction:

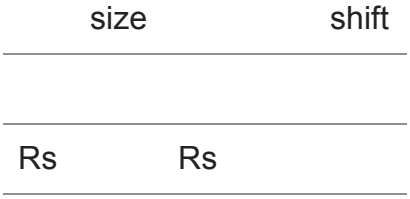


And arithmetic right shifting uses the signed bitfield extract in order to get sign-extension behavior.



Rotation can be synthesized from double-register extraction by using the rotation source as both of the source registers for extraction.

```
; rotate right by fixed amount
; extr  Rd, Rs, Rs, #shift
ror     Rd/zr, Rs/zr, #shift
```



Note that there is no “rotate with carry” instruction. The AArch32 `rrx` instruction does not exist in AArch64.[1] It would have been handy for finding the average of two unsigned integers without overflow.

The variable shifts have their own dedicated instructions.

```
; logical shift left variable
; Wd = Wn << (Wm & 31)
; Xd = Xn << (Xm & 63)
lslv    Rd/zr, Rn/zr, Rm/zr

; logical shift right variable
; Wd = Wn >> (Wm & 31), unsigned shift
; Xd = Xn >> (Xm & 63), unsigned shift
lsrv    Rd/zr, Rn/zr, Rm/zr

; arithmetic shift right variable
; Wd = Wn >> (Wm & 31), signed shift
; Xd = Xn >> (Xm & 63), signed shift
asrv    Rd/zr, Rn/zr, Rm/zr

; rotate right variable
; Rd = Rn rotated right by Rm positions
rorv    Rd/zr, Rn/zr, Rm/zr
```

Note that the shift amount is taken modulo the bit size of the operand. (This doesn’t really matter for `RORV` since rotating by the operand bit size has no effect.)

The pseudo-instructions `LSL` `LSR`, `ASR`, and `ROR` accept a register as the second input operand and convert it to the corresponding `V` instruction. This means that when writing assembly, you can just write `LSL` and let the assembler figure out which real opcode it corresponds to.

There are no `S` variants to the bit shifting instructions. They never update flags, unlike AArch32, which updated the carry with the last bit shifted out. If you want to know what bit got shifted out, you’ll have to calculate it yourself, say by shifting the same value again, but by one less position, and then inspecting the top/bottom bit (depending on the shift direction).

I have my guesses as to why the designers removed the flags behavior from these instructions: First, it removes a partial register update (flags), which creates a usually-unwanted dependency on the previous flags. Second, no major programming language gives

you access to the bit that was shifted out, so it wasn't used in practice anyway.

Exercise: Suppose there was no double-register extraction instruction or variable rotation instruction. Synthesize fixed and variable rotation from other instructions. (Answer below.)

Bonus chatter: In AArch32, the bottom 8 bits of the shift-count register were used. But in AArch64, only the bottom 5 (for 32-bit operands) or 6 (for 64-bit operands) bits are used.

Answer to exercise: You can synthesize a fixed rotation from a shift and a bitfield insertion.

```
; rotate r1 left by #imm, producing r0
                                    ; r1 = ABCDEFGH
lsl     r0, r1, #imm                ; r0 = EFGH0000
bfxil   r0, r1, #(size-imm), #imm   ; r0 = EFGHABCD
```

A variable rotation can be synthesized from a pair of shifts.

```
; rotate r1 left by r2, producing r0
; (destroys r2)
                                    ; r1 = ABCDEFGH
lslv    r0, r1, r2                  ; r0 = EFGH0000
mvn     r2, r2                      ; r2 = leftover bits
lsrv    r2, r1, r2                  ; r2 = 0000ABCD
orr     r0, r0, r2                  ; r0 = EFGHABCD
```

[1] Although it doesn't explicitly have a "rotate left through carry" instruction, you can still do it in a single instruction:

```
adcs    r0, r1, r1  ; r0 = r1 rotated left through carry
```

Raymond Chen

Follow