The AArch64 processor (aka arm64), part 6: Bitwise operations

devblogs.microsoft.com/oldnewthing/20220802-00

August 2, 2022

Raymond Chen

Bitwise logical operations are not normally particularly exciting, but for AArch64, they get exciting not so much for the operations themselves but for the immediates they can encode.

Let’s get the boring part out of the way.

```
; bitwise and with immediate
; Rd = Rn & imm
and     Rd/sp, Rn/zr, #imm

; bitwise and with shifted register
; Rd = Rn & (Rm with shift)
and     Rd/zr, Rn/zr, Rm/zr, shift

; bitwise and with immediate, set flags
; Rd = Rn & #imm, set flags
ands    Rd/zr, Rn/zr, #imm

; bitwise and with shifted register, set flags
; Rd = Rn & (Rm with shift), set flags
ands    Rd/zr, Rn/zr, Rm/zr, shift

; bitwise clear
; Rd = Rn & ~(Rm with shift)
bic     Rd/zr, Rn/zr, Rm/zr, shift

; bitwise clear, set flags
; Rd = Rn & ~(Rm with shift), set flags
bics    Rd/zr, Rn/zr, Rm/zr, shift

; bitwise or with immediate
; Rd = Rn | imm
orr     Rd/sp, Rn/zr, #imm

; bitwise or with shifted register
; Rd = Rn | (Rm with shift)
orr     Rd/zr, Rn/zr, Rm/zr, shift

; bitwise or not with shifted register
; Rd = Rn | ~(Rm with shift)
orn     Rd/zr, Rn/zr, Rm/zr, shift

; bitwise exclusive or with immediate
; Rd = Rn ^ imm
eor     Rd/sp, Rn/zr, #imm

; bitwise exclusive or with shifted register
; Rd = Rn ^ (Rm with shift)
eor     Rd/zr, Rn/zr, Rm/zr, shift

; bitwise exclusive or not with shifted register¹
; Rd = Rn ^ ~(Rm with shift)
eon     Rd/zr, Rn/zr, Rm/zr, shift
```

There are a lot of combinations here. Let’s put them in a table.

Instruction	Immediate		Shifted register	
	to Rd/sp no flags	to Rd/zr with flags	to Rd/zr no flags	to Rd/zr with flags
`AND`	•	•	•	•
`BIC`			•	•
`ORR`	•		•	
`ORN`			•	
`EOR`	•		•	
`EON`			•	

For the instructions that set flags, the N and Z flags represent the result of the operation, and the C and V flags are cleared.[2]

Stare at this table a bit and you start to see patterns.

All of the bitwise operations support a shifted register, which could be a `LSL #0` to mean "no shift". The operations that do not complement the second input operand support an immediate. (There's no need to support an immediate for the complement versions, because you can just complement the immediate.) And the `AND`-like operations are the only ones which support flags. We'll see workarounds for the lack of flags support in the other bitwise operations when we get to control transfer.

With these instructions, we can create some pseudo-instructions:[3]

```
    tst     Rn/zr, #imm             ; ands zr, Rn/zr, #imm
    tst     Rn/zr, Rm/zr, shift     ; ands zr, Rn/zr, Rm/zr, shift

    mov     Rd, #imm                ; orr  Rd, zr, #imm
    mov     Rd, Rn/zr, shift        ; orr  Rd, zr, Rn/zr, shift

    mvn     Rd, Rn/zr, shift        ; orn  Rd, zr, Rn/zr, shift
```

The `TST` pseudo-instruction performs a bitwise *and* of its arguments and sets flags, but discards the result. It's common to use a power-of-two immediate here, to test a specific bit.

The `MOV` instruction set a register equal to the value of another register or a supported immediate.

The `MVN` instruction sets a register to the bitwise inverse of another register.

Okay, so about those immediates.

The bitwise operations encode the immediates in a very strange way. If that's the sort of thing that interests you, I encourage you to read Dominik Inführ's explanation of how they are formed for the gory details.

The short version is that the immediate can encode

- a 2-bit pattern repeated 32 times,
- a 4-bit pattern repeated 16 times,
- an 8-bit pattern repeated 8 times,
- a 16-bit pattern repeated 4 times,
- a 32-bit pattern repeated 2 times, or
- a 64-bit pattern repeated 1 time.

The pattern consists of a bunch of right-justified 1's, with leading bits filled with 0's.

Finally, after concatenating the copies of the pattern, you can rotate the whole thing to the right by any amount.

For example, single bits are expressible in this format, because you can ask for a 64-bit pattern consisting of a single rightmost set bit, and then rotate that single bit into the position you like.

Conversely, all bits set except one can be generated by asking for a 64-bit pattern consisting of 63 rightmost set bits (a single clear bit in position 63), and then rotate that 0 bit into the position you like.

Interestingly, you cannot generate all ones or all zeros with this pattern. Fortunately, you don't need to. You can use *zr* for zero and the complement instruction with *zr* for ones. And operations with all ones or all zeroes can often be simplified to another instruction anyway, often avoiding a register dependency.

Missing instruction	Replacement	Note
`and Rd, Rn, #0`	`mov Rd, #0`	AND with zero is zero
`and Rd, Rn, #-1`	`mov Rd, Rn`	AND with -1 is unchanged
`orr Rd, Rn, #0`	`mov Rd, Rn`	OR with zero is unchanged
`orr Rd, Rn, #-1`	`orn Rd, zr, zr`	OR with -1 is -1
`eor Rd, Rn, #0`	`mov Rd, Rn`	EOR with zero is unchanged
`eor Rd, Rn, #-1`	`orn Rd, zr, Rn`	EOR with -1 is bitwise negation

Okay, so that's it for the bitwise logical operations. Next time, we'll look at bit shifting.

[1] The `EON` instruction is new for AArch64. AArch32 does not have this opcode.

[2] AArch32 left C and V unchanged. My guess is that AArch64 forces both bits clear in order to avoid partial flags updates, which creates unintended dependencies among instructions.

[3] AArch64 lost the `TEQ` instruction from AArch32, which I noted was of limited utility.

Raymond Chen

Follow