The AArch64 processor (aka arm64), part 15: Control transfer

devblogs.microsoft.com/oldnewthing/20220815-00

August 15, 2022

Raymond Chen

We start with the unconditional relative branch.

```
b       label       ; unconditional branch
```

The reach of the relative branch is around ±128MB. If the branch target is more than 128MB away, then the linker will modify the relative branch to point to a "trampoline", which we'll discuss a little later.

The relative branch instruction can be conditionalized on the status flags. They are the same status flags used by AArch32.

Condition	Meaning	Evaluation	Notes
`EQ`	equal	Z = 1	
`NE`	not equal	Z = 0	
`CS`	carry set	C = 1	
`HS`	high or same		unsigned greater than or equal
`CC`	carry clear	C = 0	
`LO`	low		unsigned less than
`MI`	minus	N = 1	signed negative
`PL`	plus	N = 0	signed positive or zero
`VS`	overflow set	V = 1	signed overflow
`VC`	overflow clear	V = 0	no signed overflow
`HI`	high	C = 1 and Z = 0	unsigned greater than

`LS`	low or same	C = 0 or Z = 1	unsigned less than or equal
`GE`	greater than or equal	N = V	signed greater than or equal
`LT`	less than	N ≠ V	signed less than
`GT`	greater than	Z = 0 and N = V	signed greater than
`LE`	less than or equal	Z = 1 or N ≠ V	signed less than
`AL`	always	always true	unconditional

Aside from `AL`,[1] the conditions come in pairs, and toggling the bottom bit negates the condition, which is conveniently kept in the bottom bit of the instruction, so if you want to reverse the sense of a branch, you can toggle the bottom bit. And if you want to replace the condition, you can replace the bottom nibble.

The conditions are named after the behavior that is expected if they come directly after a `CMP` instruction. For example, a `BEQ` instruction that comes directly after a `CMP` is a conditional branch that is taken if the comparison was between two equal values.

The conditional relative branches have a reach of ±1MB.

There are special conditional branch instructions for testing whether a register or bit is zero.

```
; compare and branch if zero/nonzero
cbz     Rn, label       ; branch if Rn == 0
cbnz    Rn, label       ; branch if Rn != 0

; test bit and branch if zero/nonzero
tbz     Rn, #imm, label ; branch if Rn & (1 << imm) == 0
tbnz    Rn, #imm, label ; branch if Rn & (1 << imm) != 0
```

The `CBZ` / `CBNZ` instructions have a reach of ±1MB,[2] and the `TBZ` / `TBNZ` instructions have a reach of ±32KB.

You can synthesize a "branch if negative / nonnegative" from `TBZ` and `TBNZ` by testing the sign bit.

```
; For 64-bit values, the sign bit is bit 63.
tbz     Xn, #63, label  ; branch if nonnegative
tbnz    Xn, #63, label  ; branch if negative

; For 32-bit values, the sign bit is bit 31.
tbz     Wn, #31, label  ; branch if nonnegative
tbnz    Wn, #31, label  ; branch if negative
```

The `CBZ` / `CBNZ` and `TBZ` / `TBNZ` instructions help compensate for the absence of some flags-setting bitwise operations.

```
; you want to write
eor     x0, x1, x2
bmi     negative
bne     nonzero

; alternative
eor     x0, x1, x2
tbnz    x0, #63, negative
cbnz    nonzero
```

In addition to relative jumps, we have a register indirect jump:

```
; branch to register
br      Xn/zr
```

The processor allows you to hard-code the zero register here, but that is not particularly useful unless your goal is to fault on the next cycle. (Better would be to use a permanently undefined instruction, which we'll see later. That way the crash points at the offending instruction instead of at address 0.)

Subroutine calls are performed by branching to the first instruction of the subroutine and putting the return address in the *x30* register.

```
; branch with link (can reach ±128MB)
bl      label           ; x30 = return address
                        ; execution resumes at label

; branch with link to register
blr     Xn/zr           ; x30 = return address
                        ; execution resumes at Xn
```

The branch-with-link instructions predict a subroutine call.

And of course your subroutine will probably want to return:

```
; return from subroutine
ret     Xn/zr           ; resume execution at Xn
ret                     ; resume execution at x30
```

The `RET` instruction is functionally equivalent to `BR` because they both perform a branch to an address held in a register. The difference is that `RET` predicts a subroutine return.

Okay, now about trampolines. A trampoline is a fragment of code that jumps to the final destination. To help generate the jump instruction, the code fragment is permitted to clobber the *x16* and *x17* registers, also known as *xip0* and *xip1*. Here's an example:

```
    ; original code was "bl toofar", but toofar is too far away.
    bl      toofar_trampoline

...

toofar_trampoline:
    adrp    xip0, toofar
    add     xip0, xip0, PageOffset(toofar)
    br      xip0
```

Next time, we'll look at the collection of branchless conditional execution operations.

Bonus chatter: AArch64 drops the table branch instructions which were present in AArch32. The table branch instructions were used in AArch32 primarily for dense switch statements. We'll see later how dense switch statements are handled in AArch64.

[1] There is a mystery 16th condition code, and if you follow the pattern of the existing condition codes, the missing one should be `NV` for *never*, the opposite of `AL` (always). However, if you try to use it, you'll find that it behaves the same as `AL`. This is architecturally documented behavior. So you could say that on ARM, *never* is the same as *always*.

[2] In AArch32, the `CBZ` and `CBNZ` instructions were limited to forward branches, but in AArch64 they can go both forward and backward.

Raymond Chen

Follow