The ARM processor (Thumb-2), part 12: Control transfer

devblogs.microsoft.com/oldnewthing/20210615-00

June 15, 2021

Raymond Chen

The most basic control transfer is a direct relative branch.

```
b       label       ; unconditional branch
```

The reach of the relative branch is around ±16MB, with a compact 16-bit encoding available for branch targets within 2KB.

The relative branch instruction can be conditionalized on the status flags:

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

The conditions come in pairs (aside from `AL`), and toggling the bottom bit negates the condition. For 16-bit conditional branch encoding, this maps to the bottom bit of the first byte of the instruction. For 32-bit conditional branch encoding, you toggle `0x40` in the second byte of the instruction.

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.

Four bits of instruction encoding space are lost to encode the condition, so it can reach only 1/16th as far as the unconditional branch: About ±254 bytes for the 16-bit encoding and about ±1MB for the 32-bit encoding.

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

```
cbz     Rn, label       ; branch if Rn == 0
cbnz    Rn, label       ; branch if Rn != 0
```

These are 16-bit instructions which are available only for low registers, and they are capable only of branching *forward* by up to 126 bytes.[1]

Subroutine calls are performed by branching to the first instruction of the subroutine and putting the return address in the *lr* register. This should feel familiar, for all of the other non-x86 processors we've reviewed perform subroutine linkage the same way.

```
; branch and link, stay in Thumb-2
bl      label           ; lr = next instruction + 1
                        ; execution resumes at label

; branch and link with exchange, switch to classic ARM
blx     label           ; lr = next instruction + 1
                        ; execution resumes at label
```

These instructions have a reach of approximately ±16MB.

Windows uses Thumb-2 exclusively, so you won't see the `blx` instruction used in this way. The `X` stands for "exchange", which means that it swaps between Thumb-2 and classic ARM modes.[2]

The return address is stored in *lr*, but with the bottom bit set. There's a reason for this.

Thumb-2 instructions must be halfword-aligned, and classic ARM instructions must be word-aligned. Therefore, the bottom bit of any code address is known to be zero, so the processor uses it to encode the target instruction set: If the bottom bit is clear, then execution resumes in classic ARM; if the bottom bit is set, then execution resumes in Thumb-2. Switching dynamically between classic ARM and Thumb-2 instruction sets is known as *interworking*.

Windows uses Thumb-2 exclusively, and the convention is that the bottom bit of function pointers is always set. When you look at function pointers in the debugger, they will always be *one larger* than the address itself.

```
; branch with exchange
bx      Rn              ; switch to classic ARM if Rn is even
                        ; execution resumes at Rn & ~1

; branch and link with exchange
blx     Rn              ; lr = next instruction + 1
                        ; switch to classic ARM if Rn is even
                        ; execution resumes at Rn & ~1
```

Even though the `X` instructions can switch to classic ARM, that switching feature is never used in Windows. Function pointers always have the bottom bit set, so the destination of the `BLX` is always Thumb-2.

The last branch instruction is the table-based branch:

```
; table branch byte
tbb     [Rn, Rm]            ; jump to pc + 2 * (byte at Rn + Rm)

; table branch halfword
tbh     [Rn, Rm, lsl #1]    ; jump to pc + 2 * (halfword at Rn + Rm * 2)
```

The base register points to the start of a jump table, and the second register is a byte or word index into the table. The value read from the table is then treated as a forward relative branch offset in units of halfwords.

Remember that *pc* has moved ahead four bytes when the instruction executes, so the forward branch is relative to the next instruction, not to the `TBB` or `TBH` instruction.

Since the offsets are stored in an unsigned byte or halfword, the reach of `TBB` instruction is 514 bytes, and the reach of of the `TBH` instruction is around 128KB.

One thing you might notice is that, if you assume that the bottom bit of the register is set, these two instructions are equivalent:

```
bx      Rn          ; jump to Rn
mov     pc, Rn      ; jump to Rn
```

The second version takes advantage of the fact that storing a value into the *pc* register acts as a control transfer. In practice, you won't see the `MOV` version because it takes a 32-bit encoding, whereas `BX` uses a 16-bit encoding.

Nevertheless, other variations of loading a value into *pc* are still useful:

```
mov     pc, [r0,#4] ; jump to address
pop     {pc}        ; pop return address and jump there
```

Popping a value into the instruction pointer is a common pattern. On entry to a function, you push the registers you need to preserve across the call, and on exit you pop them off. The two sets of registers line up, so that everything pops back to the original source register, *except* that you pop the old *lr* into *pc*, so that the `pop` instruction is a combination "pop registers from the stack" and "return to caller" instruction.

```
; save a bunch of registers, and the return address
push    {r3-r6,r11,lr}

...

; restore the registers, except that the return
; address goes into pc, thereby jumping there
pop     {r3-r6,r11,pc}
```

Next time, we'll look at conditional execution.

[1] The inability to branch backward with `CBNZ` explains why the sample atomic sequence we used last time uses a two-instruction sequence of `cmp r3, #0` followd by `bne`: It can't use `cbnz` because it wants to branch backward to retry the operation.

[2] This instruction was clearly named back when there were only two modes. Nowadays, naming the instruction "exchange" would be ambiguous about which of the many modes it is switching to.

Raymond Chen

Follow