The ARM processor (Thumb-2), part 19: Common patterns

devblogs.microsoft.com/oldnewthing/20210624-46

June 24, 2021

Raymond Chen

We saw some time ago how to recognize dense switch statements that use the `TBB` and `TBH` instructions. Here are some other common sequences in compiler-generated code. Note that instructions are likely to be reordered by the compiler to avoid stalls.

A call to an imported function is an indirect function call through a global function pointer:

```
movs    r0, #0          ; first parameter
movs    r1, #42         ; second parameter
ldr     r3, =|__imp__Function| ; address of global function pointer
ldr     r3, [r3]        ; load function pointer
blx     r3              ; call the function
```

A call to a virtual function is an indirect function call through a function pointer stored in the object's vtable.

```
mov     r0, r4          ; r0 = this
mov     r1, #42         ; r1 = first parameter
mov     r3, [r0]        ; r3 -> vtable
ldr     r3, [r3, #4]    ; r3 = pointer to function from vtable
blx     r3              ; call the function
```

Windows components are compiled with control flow guard (CFG), which validates indirect jump targets, making it harder for malware to redirect indirect calls to malicious payload. Calls to virtual functions go through CFG to make it harder for an attacker to manufacture a fake vtable and trick code into calling through it. A virtual function call with CFG enabled looks like this:

```
    mov     r5, [r4]        ; r5 -> vtable
    mov     r5, [r5]        ; r5 = pointer to function from vtable
    ldr     r3, =|__guard_check_call_fptr| ; address of pointer to CFG check function
    ldr     r3, [r3]        ; r3 = pointer to CFG check function
    mov     r0, r5          ; r0 = pointer to function from vtable
    blx     r3              ; check the pointer in r0

    mov     r0, r4          ; r0 = this
    mov     r1, #42         ; r1 = first parameter
    mov     r3, [r0]        ; r3 -> vtable
    blx     r5              ; call the pointer we validated
```

An important detail here is that we call indirectly through the same pointer we validated, rather than loading it from memory again. This avoids a TOCTTOU race condition, where the attacker swaps in a malicious function pointer after the old value is validated.

Another common sequence is the dense switch statement, which uses the `TBB` and `TBH` instructions.

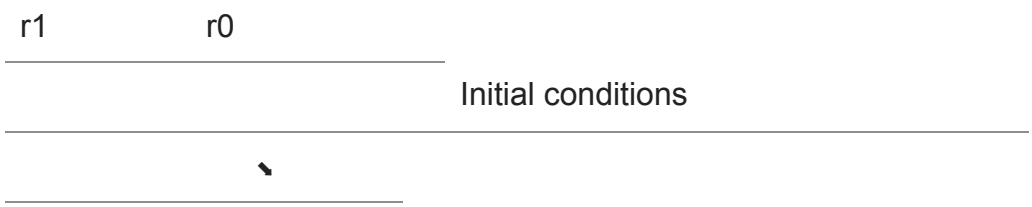
```
    cmp     r0, #8          ; beyond end of table?
    bhi     default_case    ; Y: go to the default case
    tbb     [pc, r0]        ; B: use jump table
    dcb     4, 53, 4, 53, 93, 53, 143, 172, 205
case_0:
case_2:
    ...
```

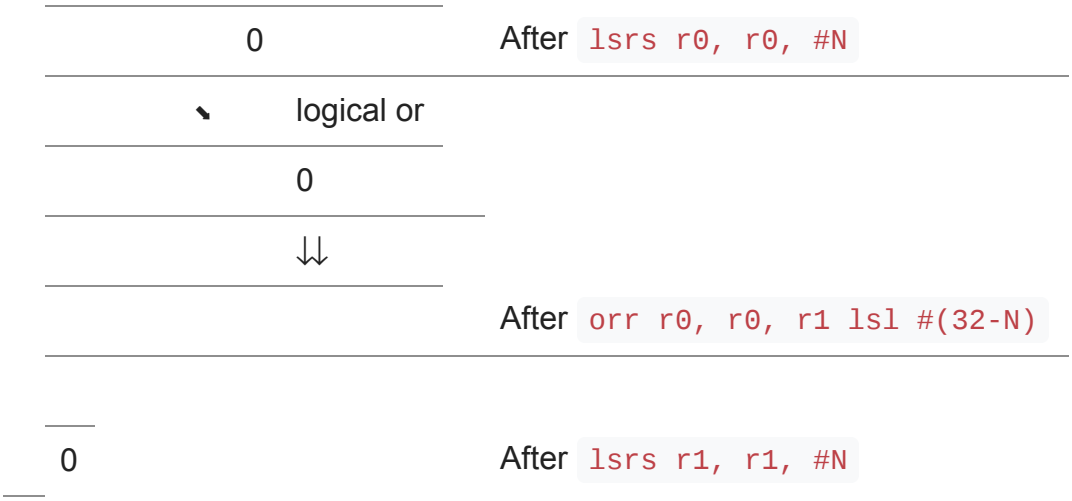
It is common to put the jump table immediately after the table branch instruction, and address it with *pc*, which has conveniently been moved forward four bytes, so it points at what would be the next instruction. In our example, the jump table is eight bytes long, so an entry of 4 means that we jump ahead 4 × 2 bytes, which takes us just past the jump table.

The barrel shifter also comes in handy when performing multiword bit shifting, since you can use the barrel shifter to isolate the bits that need to move between words.

```
    ; logical right shift doubleword in (r1,r0) by N
    lsrs    r0, r0, #N              ; logical shift right lower half
    orr     r0, r0, r1 lsl #(32-N)  ; copy low N bits 0 of r1 to high bits of r0
    lsrs    r1, r1, #N              ; logical shift right upper half
```

In pictures, we are doing this:





We shift the lower half right by *N* positions, which zeroes out the top *N* bits of the lower half. Then we use the barrel shifter to take the upper half and shift it left by 32 – *N* positions: This takes the lower *N* bits and move them to the top of the 32-bit value, clearing all the other bits. The result is then `orr` 'd into the shifted lower half, so the net effect is that the low *N* bits of the upper half are copied to the upper *N* bits of the lower half. Finally, we shift the upper half right by *N* positions.

For an arithmetic doubleword right shift, you can replace the final instruction with `asrs`. And an analogous three-instruction sequence works for multibit left shifts.

Shifting by more than 32 bits is just a matter of shifting the surviving half by *N* – 32 and either zeroing-out (for logical shifts) or sign-extending (for arithmetic right shift) the remaining bits.

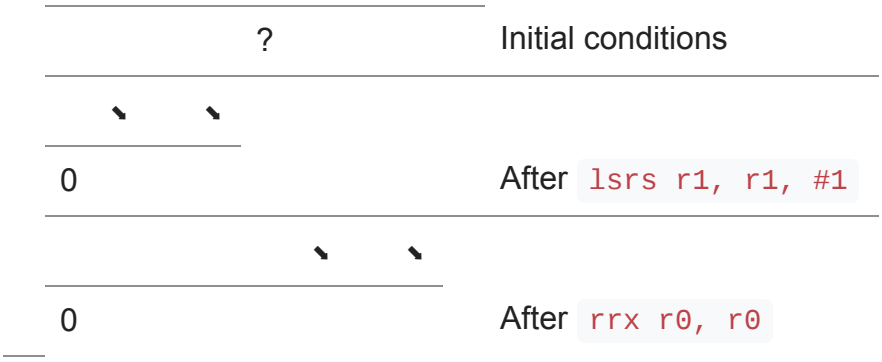
This pattern extends naturally to sizes beyond two words, though you won't see that in compiler-generated code seeing as there are no arithmetic integer types bigger than 64 bits in 32-bit Windows.

We'll wrap up the series, as is traditional, with an annotated code walkthrough of a simple function.

Bonus chatter: For the special case of shifting by one position, you can take shortcuts: Start at the opposite end and rotate the carry into the other half.

```
; single bit right shift doubleword in (r1, r0)
lsrs    r1, r1, #1          ; logical shift right upper half
rrx     r0, r0              ; rotate shifted-out bit into high bit of lower half
```

Here it is in pictures:



The trick here is that if only one bit is being shifted out, we can hold it in the carry, and then use `rrx` to shift it into the high bit of the lower half. The bottom bit of the lower half ends up in the carry, ready to be rotated into the next word (if you need to shift a large array right by one bit).

The same trick works for shifting left, using the fact that `adcs` can be used to perform a left rotate through carry.

```
; single bit left shift doubleword in (r1,r0)
adds    r0, r0, r0          ; shift left and propagate bit 31 to carry
adcs    r1, r1, r1          ; shift left and fill bottom bit with carry
```

These special-case sequences for 1-bit shifts do introduce instruction dependencies, which is bad for out-of-order execution, so compilers may avoid them for performance reasons. The results I see are inconsistent:

	Uses short version for 1-bit shift	
Compiler	**Right**	**Left**
MSVC	No	No
clang	Yes	No
gcc	No	Yes

Raymond Chen

Follow