The ARM processor (Thumb-2), part 20: Code walkthrough

devblogs.microsoft.com/oldnewthing/20210625-00

June 25, 2021

Raymond Chen

As is traditional, I wrap up the processor overview series with an annotated walkthrough of a simple function. Here's the function again:

```
extern FILE _iob[];

int fclose(FILE *stream)
{
    int result = EOF;

    if (stream->_flag & _IOSTRG) {
        stream->_flag = 0;
    } else {
        int index = stream - _iob;
        _lock_str(index);
        result = _fclose_lk(stream);
        _unlock_str(index);
    }

    return result;
}
```

Let's dive in.

This function takes a single pointer parameter, which therefore is passed in the *r0* register. No parameters are passed on the stack.

```
    push    {r3-r6,r11,lr}
```

We start by building our stack frame. From this one instruction we already learn that

- This is not a lightweight leaf function, because we are using the stack. Saving the frame pointer *r11* and return address *lr* is therefore required.
- We have one word of local variables and outbound parameters. This is inferred by the inclusion of the otherwise-garbage *r3* register. We don't actually care about the value of the *r3* register. We are pushing it for the side effect of allocating space on the stack.

- We need three additional registers: *r4*, *r5*, and *r6*.

```
    add     r11, sp, #0x10      ; link into stack frame chain
```

The next step in the standard prologue is to point the *r11* register at the place where we saved the previous *r11* register, in order to maintain the stack frame chain.

```
    mov     r5, r0              ; r5 = stream
```

We save the stream pointer in a non-volatile register for safekeeping.

```
;   int result = EOF;
;   if (stream->_flag & _IOSTRG) {

    ldr     r3, [r5,#0xC]       ; r3 = stream->_flag
    mvn     r6, #0              ; r6 "result" = -1
    tst     r3, #0x40           ; Q: Is _IOSTRG set?
    beq     notstring           ; N: Then need to flush for real
```

The compiler interleaved the initialization of the *result* variable (which is evidently being kept in register *r6*) with the test of the `_flags` member.

Initializing *result* is done by moving `~0`, which is the same as `0xFFFFFFFF` or `-1`.

Testing the `_IOSTRG` bit is done by loading the flags into the *r3* register (a scratch register) and using the `TST` instruction, which sets the flags based on the result of a bitwise AND operation. If the flag is clear, then the result is zero (“equal”), and the jump is taken. If the flag is set, then we fall through.

```
;   stream->_flag = 0;

    movs    r3, #0              ; r3 = 0
    str     r3, [r5,#0xC]       ; stream->_flag = 0
    b       done                ; end of "true" branch
```

If the flag is clear, then we enter the “true” branch of the `if` statement, which sets the `_flag` to zero. We cannot move a constant directly into memory, so we first load the constant in to a scratch register (*r3*) and store the register to memory.

Note that we use a `MOVS` instruction, which sets flags, even though we don’t care about the flags. That’s because the 8-bit immediate `MOVS` instruction has a compact 16-bit encoding, whereas the corresponding `MOV` instruction uses a 32-bit encoding, so switching to `MOVS` reduces code size.

```
notstring:
;   int index = stream - _iob;

    ldr     r3, =|_iob|         ; r3 = address of _iob
    subs    r4, r5, r3          ; r4 = stream - iob (byte offset)
    asrs    r0, r4, #4          ; r0 = r4 / 16 (convert to index)
```

We use the literal pool version of the `LDR` pseudo-instruction to load the address of the `_iob` array from the literal pool into a scratch register *r3*. We subtract that from the *stream* variable, producing the byte offset into the preserved register *r4*. Shifting that right by 4 is the same as dividing by 16, which produces the index into the *r0* register.

```
;   _lock_str(index);

    bl      |_lock_str|
```

The *r0* register is exactly where we pass the *index* parameter to the `lock_str` function, so we're all set to call it.

```
;   result = _fclose_lk(stream);

    mov     r0, r5              ; r0 = stream
    bl      |_fclose_lk|        ; _fclose_lk(stream)
    mov     r6, r0              ; save result
```

Next comes another function call, this time to close the stream. We put the first (and only) parameter into *r0* and call the function. The result comes back in *r0*, and we save it in *r6* so we can return it when we're done.

```
;   _unlock_str(index);

    asrs    r0, r4, #4          ; r0 = r4 / 16 (convert to index)
    bl      |_unlock_str|       ; _unlock_str(index)
```

To call `_unlock_str`, we recalculate the index from the byte offset (still in *r4*, since *r4* is a preserved register) and put the index into *r0* so we can call `_unlock_str`.

It may seem odd to recalculate the index from the byte offset. Why not just save the index the first time?

The reason is that `mov r0, r4` and `asrs r0, r4, #4` are the same size: They both use 16-bit encoding. Recalculating the value takes the same number of code bytes as copying it, and it avoids having to save the index anywhere, thereby saving two bytes. Thanks to the barrel shifter (which the ARM is very proud of, in case you have forgotten), shifting a register is just as fast as copying it.

We now fall through to the end of the function.

```
done:

;   return result;

    mov     r0, r6              ; return result (r6)
```

The function return value goes into *r0*, so we copy it there from *r6*.

```
    pop     {r3-r6,r11,pc}
```

For this function, we can pack the the function epilogue into just one instruction: Popping *r3* cleans up our local variables, popping *r4* through *r6* restores the saved registers, popping *r11* unlinks the current stack frame from the stack frame chain, and popping the inbound return address into *pc* transfers control to the return address.

That's the end of the function, but we're not done yet!

```
    __debugbreak                ; recover word alignment

    dcd     |_iob|
```

We still have the matter of the literal pool we used in the `ldr r3, =|_iob|` pseudo-instruction. That pseudo-instruction turns into the instruction

```
    ldr     r3, [pc, #...]      ; load register from memory
```

where the `#...` is the offset to the desired literal. When you use the *pc* register as a base index, the value is rounded down to the nearest multiple of four, and the offset must also be a multiple of four. This means that the value must be at a word-aligned address. The unreachable `__debugbreak` instruction at the end of the function is just padding so that the `|_iob|` literal can be placed on a word boundary.

So there we have it, our whirlwind tour of the ARM processor in Thumb-2 mode. I don't know about you, but I'm exhausted.

[1] Commenter Neil Rashbrook notes that stack space reserved by pushing the *r3* register is never used. It exists only to satisfy the requirement that the stack be 8-byte aligned.

Raymond Chen

Follow