The ARM processor (Thumb-2), part 2: Differences between classic ARM and Thumb-2

devblogs.microsoft.com/oldnewthing/20210601-00

June 1, 2021

Raymond Chen

As I noted in the introduction, classic ARM encodes instructions as as 32-bit values which must reside on a word boundary. However, Windows uses the ARM processor exclusively in a mode known as Thumb-2, which uses a variable-sized encoding format: Instructions can be 16-bit or 32-bit, with the goal of providing more compact 16-bit encodings for the most common instructions.

ARM computations are typically three-register, with an output register and two input registers. Let consider the `ADCS` instruction, which is "add with carry and set flags". If you use the fully-general form, it will cost you a 32-bit instruction.

```
; 32-bit instruction
adcs    r0, r1, r2      ; r0 = r1 + r2 + carry, set flags
```

But if you make the output register equal to the first input register, *and* if all the registers are in the range *r0* through *r7*, then a compact 16-bit encoding often becomes available.

```
; 16-bit instruction
adcs    r0, r0, r2      ; r0 = r0 + r2 + cary, set flags
```

The registers *r0* through *r7* are known as *low registers*, and the ones from *r8* through *r15* are called *high registers*. So we can say that a compact 16-bit encoding for `ADCS` becomes available if you can reduce the instruction to *two low registers*.

Even reducing to two low registers may not be enough. For example, the `ADC` instruction (add with carry, but without setting flags) has no 16-bit encoding. It will always require a 32-bit instruction. This means that code generation may end up picking the `ADCS` instruction instead of `ADC` even though you might naïvely think it's being wasteful: Why are you asking the CPU to set flags that you aren't interested in? Reason: Because it reduce code size.

The Thumb-2 instruction encodings are rather messy in order to squeeze as many useful instructions into the 16-bit space. For example, the `ADD` instruction has a 16-bit encoding for the three-register version, provided all of the registers are low. The precise conditions under which an instruction supports a 16-bit encoding vary wildly from instruction to instruction. The designers sacrificed decoding simplicity for code density.

Remember, the focus of this series is knowing enough to read compiler-generated assembly, not to be able to write your own from scratch. The point of this discussion is not to teach you about which instructions have 16-bit encodings, but rather to point out that you may see something unusual in the code generation due to the desire to avoid 32-bit instructions if a 16-bit alternative is available.

In order to free up instruction encoding space, operations on the *sp* and *pc* registers are more limited. For example, you can add to and subtract from the *sp* register, but you can't, say, rotate the *sp* register left by 5. Since the stack pointer is architectural in Thumb-2, there isn't a common real-world scenario where you would need to do weird arithmetic on the stack pointer. Similarly, most arithmetic operations on the *pc* register are prohibited. The encodings that correspond to all of these prohibited operations have either been re-used to encode other instructions, or remain reserved for future use.

Another significant place where Thumb-2 differs from classic ARM is in conditional execution. In classic ARM, nearly every instruction can be made conditional: Appending a condition code to the mnemonic makes the instruction execute only if the condition is satisfied. (We'll learn more about condition codes later.) One of the condition codes is called `AL` (always), and internally, an unconditional instruction is just a conditional instruction with the `AL` condition code. There are 16 condition codes, which means that four bits of every classic ARM instruction is devoted to the condition.

Thumb-2 can't afford to give up four bits in its instruction encoding for conditional execution, so it externalized the condition with the if-then instruction (`IT`) which acts like a conditional prefix to the next instruction:

```
;   classic ARM

    addge   r0, r1, r2  ; r0 = r1 + r2 if ge condition is set

;   Thumb-2

    it ge               ; next instruction executes if ge
    addge   r0, r1, r2  ; r0 = r1 + r2
```

In the instruction stream, the instruction after the `IT` is just a plain `ADD` instruction, but the assembler requires you to write `ADDGE` as a double-check. Conversely, the assembler checks that your conditional instruction is preceded by a matching `IT` .

At run time, if the condition in the `IT` instruction is not met, then the next instruction is ignored.

The if-then instruction can conditionalize up to four instructions. You specify how many instructions you want to conditionalize by adding up to three `T` or `E` suffixes to the opcode, indicating whether that instruction should be executed if the condition is true or false. (The `E` stands for *else*.)

```
;   Thumb-2

    ite ge              ; if-then-else
    strge   r0, [r2]    ; store r0 to [r2] if ge
    strlt   r1, [r2]    ; store r1 to [r2] if not ge

;   classic ARM equivalent

    strge   r0, [r2]    ; store r0 to [r2] if ge
    strlt   r1, [r2]    ; store r1 to [r2] if lt
```

There are constraints on what you can do inside an if-then block: You cannot transfer into the middle of an if-then block,[1] and only the last instruction in the block can be a control transfer or an instruction that modifies flags.

The Windows ABI imposes further restrictions on the use of the `IT` instruction. Even though the processor lets you conditionalize up to four instructions, Windows allows you to conditionalize only one instruction, and it must be one of a limited set of 16-bit instructions.[2] You can read the details if that's the sort of thing that turns you on.

Next time, we'll look at the addressing modes.

[1] Why does the processor care if you jump into the middle of an `IT` sequence? It's not like it looks backward in the instruction stream to see if an `IT` instruction came ahead of the instruction you jumped to. And who knows, maybe some data that came before the instruction just *happens* to look like an `IT` instruction if disassembled as code. What's the reason for this rule, and how does it work?

My guess is that this rule is to simplify instruction caching. With this rule, it means that the processor can cache the decoded instruction, along with its conditions. If you could jump into an `IT` block, then the processor would have to re-decode the condition because the condition would be different depending on whether execution fell through the preceding `IT` instruction or whether execution jumped directly to the controlled statement.

Therefore, the rule is not so much that jumping into the middle of an `IT` block is prohibited, but rather that an instruction must always be executed in the same `IT` context: either always controlled by an encompassing `IT` instruction or never controlled by it.

[2] It's not like the processor notices that you broke the rule. What actually happens is that the instructions beyond the first controlled instruction may find themselves executed incorrectly if a hardware interrupt occurs while the CPU is in the middle of the `IT` block.

Raymond Chen

Follow