The ARM processor (Thumb-2), part 15: Miscellaneous instructions

devblogs.microsoft.com/oldnewthing/20210618-00

June 18, 2021

Raymond Chen

There are far more ARM instructions than I'm going to cover here. I've skipped over the floating point instructions, the SIMD instructions, and some other specialty instructions that I haven't yet seen come out of the compiler.

Here are a few that are still interesting, even if I haven't seen the compiler generate them.

```
; count leading zeroes (high order bits)
clz     Rd, Rm          ; Rd = number of leading zeroes in Rm

; reverse bits
rbit    Rd, Rm          ; Rd = Rm bitwise reversed

; reverse bytes
rev     Rd, Rm          ; Rd = Rm bytewise reversed

; reverse bytes in each halfword
rev16   Rd, Rm          ; Rd[31:24] = Rm[23:16]
                        ; Rd[23:16] = Rm[31:24]
                        ; Rd[15: 8] = Rm[ 7: 0]
                        ; Rd[ 7: 0] = Rm[15: 8]

; reverse bytes in lower halfword and sign extend
revsh   Rd, Rm          ; Rd[31:8] = Rm[ 7:0] sign extended
                        ; Rd[ 7:0] = Rm[15:8]
```

A few miscellaneous bit-fiddling instructions. The reversal instructions are primarily for changing data endianness.

The next few instructions provide multiprocessing hints.

```
; yield to other threads
yield

; wait for interrupt
wfi
```

The `YIELD` instruction is a hint to multi-threading processors that the current thread should be de-prioritized in favor of other threads. You typically see this instruction dropped into spin loops, via the intrinsic `__yield()`.

The `WFI` instruction instructs the processor to go into a low-power state until an interrupt occurs. There are other instructions related to "events" which I won't bother going into.

The next few instructions are for communicating with the operating system:

```
svc     #imm8       ; system call
bkpt    #imm8       ; software breakpoint
udf     #imm8       ; undefined opcode[1]
```

The system call and breakpoint instructions both carry an 8-bit immediate that the operating system can choose to use for whatever purpose it desires. The breakpoint instruction breaks the rules and always executes even if an encompassing `IT` instruction would normally cause it to be ignored. In other words, `bkpt` overrides `IT`.

The undefined opcode is a block of 256 instructions from `0xde00` through `0xdeff` that are architecturally set aside as undefined instructions and which will not be given meaning in future versions of the processor.

But just because the processor leaves them undefined doesn't mean that operating system can't give them special meaning. Windows defines custom artificial instructions in the undefined space.[2]

```
__debugbreak            ; udf #0xFE
__debugservice          ; udf #0xFD
__assertfail            ; udf #0xFC
__fastfail              ; udf #0xFB
__rdpmccntr64           ; udf #0xFA
__brkdiv0               ; udf #0xF9
```

Most of these are special ways of manually generating specific exceptions.

Opcode	Exception	Notes
`__debugbreak`	`STATUS_BREAKPOINT`	The "real" breakpoint instruction.
`__debugservice`	`STATUS_BREAKPOINT`	Communicate with debugger, *r12* is function code.
`__assertfail`	`STATUS_ASSERTION_FAILURE`	
`__fastfail`	`STATUS_STACK_BUFFER_OVERRUN`	Misleadingly-named.

`__brkdiv0`	`STATUS_INTEGER_DIVIDE_BY_ZERO`	

The `__brkdiv0` instruction is emitted by the compiler if it detects a zero denominator.

```
    cbnz    r0, @F          ; jump if denominator is nonzero
    __brkdiv0               ; oops: manually raise div0 exception
@@: bl      __rt_sdiv       ; software divide/remainder
                            ; (r0, r1) = (r1 ÷ r0, r1 mod r0)
```

The last artificial instruction is `__rdpmccntr64`, which reads a 64-bit cycle counter. This special instruction has a dedicated fast path through the trap handler, so it can produce the result in around 60 cycles.

There is also an instruction to access coprocessor registers.

```
    ; move register from coprocessor
    mrc (a bunch of stuff)
```

The coprocessor registers are encoded in a totally wacky way. There's no point learning what each of the values means. All that matters is that they represent the register you want to read.

There are a few coprocessor registers named *software thread ID register* which are not used by the processor, but are provided with the intention that operating systems use them to record per-thread information. The two available from user mode are named `TPIDRURW` and `TPIDRURO`; the first is read-write and the second is read-only. Windows uses `TPIDRURW` to hold the thread information.

And of course, we have this guy:

```
    nop
```

Actually, there are two of this guy, a 16-bit `NOP` and a 32-bit `NOP`. The `NOP` instruction does nothing but occupy space. Use it to pad code to meet alignment requirements, but do not use it for timing because processors are allowed to optimize it out, or even run *faster*.

Now that we have the basic instruction set under our belt, we'll look at the calling convention next time.

Bonus chatter: Why doesn't Windows use `udf #0xff`? The gcc toolchain uses `udf #0xff` as its "We should never get here" trap instruction. Putting an artificial instruction there would cause such a program to continue executing after it thought it had triggered a fatal exception.

[1] Although the ARM documentation provides the `udf` mnemonic for the undefined instruction, not all assemblers recognize it, so you may be forced to encode the hex value directly into your code if that's what you want.

[2] I don't know why Windows chose the `udf` space for these artificial opcodes instead of using the `svc` space. Maybe there's some fine print in the processor manual that makes `svc` unsuitable for this sort of thing. We know that `bkpt` is a bad choice for an artificial opcode because `bkpt` executes even if an encompassing `IT` instruction would have skipped it.

Then again, use of `udf` to create artificial instructions is explicitly listed in the processor architecture manual as a valid use of the `udf` instruction, so at least it's not breaking any unwritten rules.

Raymond Chen

Follow