The ARM processor (Thumb-2), part 11: Atomic access and barriers

devblogs.microsoft.com/oldnewthing/20210614-00

June 14, 2021

Raymond Chen

On the ARM processor, atomic operations are implemented in terms of a load-locked/store-conditional pair of instructions.

```
LDREX   Rd, [Rn, #imm8]     ; load word from [Rn, #imm8] and acquire exclusively
STREX   Rd, Rm, [Rn, #imm8] ; store Rm to [Rn, #imm8] if exclusively held
                            ; Rd = 0 on success or 1 on failure

; also LDREXB, LDREXH, LDREXD
;      STREXB, STREXH, STREXD

CLREX                       ; release exclusive lock
```

The `LDREX` instruction loads a word from the specified address and takes an exclusive lock on the memory. This exclusive lock is broken if any other processor writes to the same address, or if the lock is explicitly cleared. The granularity of the lock is permitted to be as coarse as 2KB.

The `STREX` instruction writes the value *Rm* to *Rn* provided the exclusive lock has not been lost. The *Rd* register is set to 0 if the write succeeded, or 1 if the write failed. The *Rd* register may not be the same register as *Rm*.

The `STREX` is permitted to early-out and return failure due to a lost lock before checking whether the memory at *Rn* is writable.

The `LDREX` and `STREX` instructions support only offset addressing with an unsigned 8-bit offset. (An offset of zero is assumed if none is provided.) No pre-indexing or post-indexing allowed.

There are also byte, word, and doubleword versions of this pair of instructions. For best results, use the `STREX` variant that matches the `LDREX` variant, and with the same address.

You can explicitly abandon a lock obtained by one of the `LDREX` instructions by issuing a `CLREX` instruction. This is used primarily in kernel mode to ensure that interrupts and context switches cause the lock to be lost: If the user-mode code is interrupted between the `LDREX` and the subsequent `STREX`, you want to make sure the `STREX` fails, rather than accidentally succeeding because it's writing to an address that coincidentally matches a previous `LDREX` from the outgoing thread or interrupt.

The atomic memory access instructions require aligned memory. Relaxing alignment enforcement doesn't help here. Not that you expect it to: How can the kernel emulate a misaligned lock?

The atomic memory operations are frequently coupled with synchronization primitives. The ARM processor has a rather weak memory model, so memory barriers are essential in proper multithreaded code.

```
    DMB     ish     ; data memory barrier
    DSB     ish     ; data synchronization barrier
    ISB     sy      ; instruction synchronization barrier
```

The data memory barrier ensures that all preceding writes are issued before any subsequent memory operations (including speculative memory access). In acquire/release terms, it is a full barrier. The instruction does not stall execution; it just tells the memory controller to preserve externally-visible ordering. This is probably the only barrier you will ever seen in user-mode code.

The data synchronization barrier is a data memory barrier, but with the additional behavior of stalling until all outstanding writes have completed. This is typically used during context switches.

The instruction synchronization barrier flushes instruction prefetch. This is typically used if you have generated new code, say by jitting it or paging it in from disk.

All of the barrier instructions take a parameter known as the *sychronization domain*. In practice, they will be the values I gave in the examples above.

A typical atomic sequence, complete with memory barriers, looks like this:

```
    dmb     ish             ; memory barrier

@@: ldrex   r2, [r0]        ; load r2 from [r0] and lock

    ; calculate new value - in this example, we increment
    adds    r2, r2, #1      ; increment it

    strex   r3, r2, [r0]    ; store if lock is still held
    cmp     r3, #0          ; did it succeed?
    bne     @B              ; N: try again

    dmb     ish             ; memory barrier
```

Finally, we have some instructions that provide hints to the processor about future memory usage:[2]

```
    PLD     [Rn, #imm]      ; preload data
    PLDW    [Rn, #imm]      ; preload data with intent to write
    PLI     [Rn, #imm]      ; preload instructions
```

Processors are not required to honor these instructions and may treat them as nop. (Pre-index and post-index are not supported, so you don't have to worry about accidentally nop'ing out the write-back.) If the address being prefetched is not valid, the request is ignored.

Okay, enough about memory. Next time, we'll look at control transfer instructions.

Bonus chatter: Classic ARM also contains two deprecated pseudo-atomic instructions:

```
    ; swap
    swp     Rt, Rt2, [Rn]   ; temp = [Rn]
                            ; [Rn] = Rt2
                            ; Rt = temp

    ; swap byte
    swpb    Rt, Rt2, [Rn]   ; temp = byte at [Rn]
                            ; byte at [Rn] = Rt2
                            ; Rt = temp (zero-extended)
```

These are pseudo-atomic instructions because the processor promises that it will not split the load and store, but only if no TLB eviction occurs, and it makes no promises about what other processors or devices may see.

These instructions are formally deprecated by ARM, and operating systems are permitted to disable them outright. Windows disables them, which is redundant because the instructions aren't available in Thumb-2 mode anyway. I guess Windows wants to make extra sure you don't use them.

[1] Even if alignment enforcement is relaxed, you will still get an alignment exception for misaligned doubleword access or any instruction that reads or writes multiple registers.

[2] Internally, these instructions reuse the encodings for loading partial values into *pc*, something you would never do in sane code. This is an example of how Thumb-2 disallows certain operations with *pc* and reuses the instruction encodings for other purposes.

Instruction	Encoded as if
`PLD  [...]`	`LDRB  pc, [...]`
`PLDW [...]`	`LDRH  pc, [...]`
`PLI  [...]`	`LDRSB pc, [...]`

Raymond Chen

Follow