The SuperH-3, part 11: Atomic operations

devblogs.microsoft.com/oldnewthing/20190819-00

August 19, 2019

Raymond Chen

The SH-3 has a very limited number of read-modify-write operations. To recap:

```
AND.B #imm, @(r0, GBR)  ; @(r0 + gbr) &= 8-bit immediate
OR.B  #imm, @(r0, GBR)  ; @(r0 + gbr) |= 8-bit immediate
XOR.B #imm, @(r0, GBR)  ; @(r0 + gbr) ^= 8-bit immediate
TAS.B @Rn               ; T = (@Rn == 0), @Rn |= 0x80
```

These instructions are "atomic" in the sense that they occur within a single instruction and are hence non-interruptible. Technically, only the last one is truly atomic in the sense that the processor holds the data bus locked for the duration of the instruction.

Let's not quibble about such details. Let's just say we're looking for non-interruptible instructions.

The SH-3 does not support symmetric multiprocessing, so we don't have to worry about competing accesses from other main processors (although there may be competing accesses from coprocessors or hardware devices). But how are we going to build atomic increment, decrement, and exchange out of these guys?

Let's be honest. We can't.

We'll have to fake it.

Windows CE takes a different approach from how Windows 98 created atomic operations on a processor that didn't support them.

On Windows CE, the kernel is in cahoots with the implementations of the interlocked operations. If it discovers that it interrupted a special uninterruptible sequence, it resets the program counter back to the start of the uninterruptible sequence before allowing user mode to resume.[1] In this way, the kernel manufactures multi-instruction uninterruptible sequences.

These sequences have to be carefully written so that they are restartable. This means that they cannot mutate any input parameters, and there are no memory updates until the final instruction in the sequence.

For example, we could try to implement our fake `InterlockedIncrement` like this:

```
; on entry:
;   r4 = address to increment
; on exit:
;   r0 = incremented value

InterlockedIncrement:
    mov.l   @r4, r0     ; load current value    ; (1)
    add     #1, r0      ; increment it          ; (2)
    mov.l   r0, @r4     ; store updated value   ; (3)
    rts                 ; return                ; (4)
```

We load the current value from memory, add 1, store it back, and return. If this sequence is interrupt at any point, the kernel moves the program counter back to the first instruction and restarts the entire operation.

Let's walk through the possible interrupts.

- If interrupted prior to the first instruction, then moving the program counter back to the first instruction has no effect because that's where it already was. So no problems there.
- If interrupted prior to the second instruction, then we will perform the `mov.l @r4, r0` a second time. Since we haven't changed *r4*, this will read the desired memory location. It's a redundant read, but at least it's not harmful.
- If interrupted prior to the third instruction, then we will reload and re-increment the existing value. Again, since we haven't changed *r4*, this will read the correct location.
- If interrupted prior to the fourth instruction, then we're in trouble. We have already written the updated value back to memory, and restarting the operation will increment it a second time! **This code is broken.**

Aha, but we forgot about the branch delay slot of the `rts` instruction, and in fact it's the branch delay slot that provides our escape hatch: Move the final store *into the branch delay slot*.

```
; on entry:
;   r4 = address to increment
; on exit:
;   r0 = incremented value

InterlockedIncrement:
    mov.l   @r4, r0     ; load current value    ; (1)
    add     #1, r0      ; increment it          ; (2)
    rts                 ; return               ; (3)
    mov.l   r0, @r4     ; store updated value   ; (4)
```

Okay, let's run our analysis again.

- If interrupted prior to the first instruction, our analysis from above is still correct.
- If interrupted prior to the second instruction, our analysis from above is still correct.
- If interrupted prior to the third instruction, our analysis from above is still correct.
- An interrupt between the third and fourth instruction is not possible because the processor disables interrupts between a delayed branch instruction and its delay slot. But if an exception occurred (say, because the memory was copy-on-write), we can safely restart the operation because we haven't modified *r4* or the value in memory at *r4*.[2]
- If interrupted after the fourth instruction, then the program counter isn't in our special code region, so the kernel won't restart the sequence.

The branch delay slot saved us!

You never thought you'd see the day when you'd be thankful for a branch delay slot.

The kernel puts these special uninterruptible sequences in a contiguous region of memory. Let's say that it starts each special uninterruptible sequence on a 16-byte boundary. This means that the "special uninterruptible sequence detector" can go something like this:

```
    mov.l   @(usermode_pc), r0          ; see where we're returning to
    mov.l   #start_of_sequences, r1     ; the start of our special sequences
    mov     #length_of_sequences, r2    ; the size in bytes
    sub     r1, r0
    cmp/hs  r0, r2                      ; is it in the magic region?
    bf      fixme                       ; Y: then go fix it
return_to_user_mode:
    ... continue as usual ...

fixme:
    mov     #-15, r2                    ; mask out the bottom 4 bits
    and     r2, r0                      ; to go back to start of special sequence
    add     r1, r0                      ; convert from offset back to address
    bra     return_to_user_mode
    mov.l   r0, @(usermode_pc)          ; update user mode program counter
```

This is not actually how it goes, but it gives you the basic idea. In reality, the special uninterruptible sequences start on 8-byte boundaries, in order to pack them more tightly. Sequences that are longer than 4 instructions need to be arranged so that every 8 bytes is a valid restart point. I just used 16-byte sequences to make the explanation simpler.

For example, `InterlockedCompareExchange` really went like this:

```
; on entry:
;   r4 = address of value to test
;   r5 = replacement value (if current value matches expected value)
;   r6 = expected value
; on exit:
;   r0 = previous value

InterlockedCompareExchange:
    mov.l   @r4, r0     ; load current value
    cmp/eq  r0, r6      ; is it the expected value?
    bf      nope        ; Nope, just return current value
    mov.l   r5, @r4     ; Store the replacement value
nope:
    rts
    nop
```

There is a second restart point after four instructions, at the `rts`, and it's okay to restart there because the operation is complete. All we're doing is returning to our caller.

This trick for creating restartable multi-instruction sequences was not unique to the SH-3. Windows CE employed it to synthesize pseudo-atomic operations for other processors, too.

One curious side effect of this design for restartable multi-instruction sequences is that you can't debug them! If you try to single-step through these multi-instruction sequences, you'll get stuck on the first instruction: The breakpoint will fire, and the kernel will reset the program counter back to the first instruction.

Next time, we'll look at the Windows CE calling convention.

Bonus chatter: The SH-4A processor added load-locked and store-conditional instructions, bringing it in line with other RISC processors.

```
    MOVLI.L @Rm,r0          ; Load from @Rm, remember lock
    MOVCO.L r0,@Rn          ; Store to @Rn provided lock is still valid
                            ; T = 1 if store succeeded, 0 if failed
```

Bonus chatter 2: What about the TEB? Where does Windows keep per-thread information?

Turn out this is easier than it sounds. The SH-3 doesn't support symmetric multiprocessing, so there is only one processor, which therefore can be executing only one thread at a time. A pointer to the per-thread information is stored at a fixed location, and that pointer is updated at each thread switch.

[1] Fast Mutual Exclusion for Uniprocessors. Brian Bershad, David Redell, and John Ellis, Proceedings of the fifth international conference on Architectural support for programming languages and operating systems, 1992.

[2] Suppose an exception occurs in the delay slot because the memory isn't writable, and the exception handler fixes the problem (by making the memory writable on demand). Resuming execution will rewind the instruction pointer back to the start of the sequence because the memory value may have changed as part of handling the exception.

Raymond Chen

Follow