The Alpha AXP, part 9: The memory model and atomic memory operations

devblogs.microsoft.com/oldnewthing/20170817-00

August 17, 2017

Raymond Chen

The Alpha AXP has a notoriously weak memory model. When a processor writes to memory, the result becomes visible to other processors eventually, but there are very few constraints beyond that.

For example, writes can become visible out of order. One processor writes a value to a location, and then writes a value to another location, and another processor can observe the second write without the first. Similarly, reads can complete out of order. One processor reads a value from a location, then reads from another location, and the result could be that the second read happens before the first.[1]

Assume that memory locations *x* and *y* are both initially zero. The following sequence of operations is valid.

Processor 1	Processor 2
write 1 to *x*	read *y* yields 1
	`MB` (memory barrier)
write 1 to *y*	read *x* yields 0

The memory barrier instruction `MB` instructs the processor to make all previous loads and stores complete to memory before starting any new loads and stores. However, it doesn't force other processors to do anything; other processors can still complete their memory operations out of order, and that's what happened in the above example.

Similarly, the following sequence is also legal:

Processor 1	Processor 2
write 1 to *x*	read *y* yields 1

`MB` (memory barrier)	
write 1 to *y*	read *x* yields 0

This is also legal because the memory barrier on processor 1 ensures that the value of *x* gets updated before the value of *y*, but it doesn't prevent processor 2 from performing the reads out of order.

In order to prevent *x* and *y* from appearing to be updated out of order, *both* sides need to issue memory barriers. Processor 1 needs a memory barrier to ensure that the write to *x* happens before the write to *y*, and processor 2 needs a memory barrier to ensure that the read from *y* happens before the read from *x*.

Okay, onward to atomic operations.

Performing atomic operations on memory requires the help of two new pairs of instructions:

```
    LDL_L   Ra, disp16(Rb)  ; load locked
    LDQ_L   Ra, disp16(Rb)

    STL_C   Ra, disp16(Rb)  ; store conditional
    STQ_C   Ra, disp16(Rb)
```

The *load locked* instruction performs a traditional read from memory, but also sets the *lock_flag* and memorizes the physical address in *phys_locked*. The processor monitors for any changes to that physical address from any processor, and if a change is detected,[2] the *lock_flag* is cleared.

The *lock_flag* is also cleared by a variety of other conditions, most notably when the processor returns from kernel mode back to user mode. This means that any hardware interrupt or trap (such as a page fault, or executing an emulated instruction) will clear the *lock_flag*. It is recommended that operating systems allow at least 40 instructions to execute between timer interrupts.

You can later do a *store conditional* operation which will store a value to a memory address, provided the *lock_flag* is still set. If so, then the source register is set to 1. If not, then the source register is set to 0 and the memory is left unmodified. Regardless of the result, the *lock_flag* is cleared.

A typical atomic increment looks like this:

```
retry:
    LDL_L   t1, (t0)        ; load locked
    ADDL    t1, #1, t1      ; increment value
    STL_C   t1, (t0)        ; store conditional
                            ; t1 = 1 if store was successful
    BEQ     t1, failed      ; jump if store failed
    ... continue execution ...

failed:
    BR      zero, retry     ; try again
```

In the case where the store failed, we jump forward, and then back. Recall that conditional jumps backward are predicted taken, and conditional jumps forward are predicted not taken. If we had simply jumped backward on failure, then the processor would have a branch prediction miss in the common case that there is no contention.

Note that the above sequence does not impose any memory ordering. In practice, you will see a `MB` before and/or after the atomic sequence in order to enforce acquire and/or release semantics.

There are a number of practical rules regarding the `LDx_L` and `STx_C` instructions. The most important ones are these:

- The `STx_C` should be to the same address as the most recently preceding `LDx_L`. This isn't a problem in practice because storing back to the location of the previous load is the intended use of the instructions.[3]
- The processor may lose track of your `LDx_L` if you perform any memory access other than a matching `STx_C`, or if you perform a branch instruction, or if you trigger a trap (such as executing an emulated instruction), or if you execute more than 20 instructions after the `LDx_L`.

Although each `STx_C` should be preceded by a matching `LDx_C`, it is legal to perform a `LDx_L` with no matching `STx_C`. This can happen with conditional interlocked operations, where you discover after the `LDx_L` that the condition is not satisfied and you abandon the interlocked operation.

The second rule says basically that the state created by the `LDx_L` instruction is ephemeral. After performing the `LDx_L` instruction, do as little work as possible to determine what value you want to store, and then store it right away. You are not allowed to take any branches, but `CMOVcc` is okay.

The requirement that you get around to the `STx_C` within 20 instructions is a consequence of the requirement on operating systems that they allow 40 instructions to execute between timer interrupts.

Next time, we'll do a little exercise based on what we've learned so far.

[1] Mind you, out-of-order reads are pretty common on all architectures. Store-to-load forwarding means that a speculated read operation to speculatively-written memory can complete before a read operation that occurred notionally earlier in the instruction stream. However, as Fabian Giesen notes, the x86 has extra logic to avoid getting caught doing so!

[2] The architecture permits implementations to be a little sloppy with the change detection. In particular, any modification within 128 bytes of the locked address is permitted to clear the *lock_flag*. This means that targets of atomic operations should be at least 128 bytes apart in order to minimize the likelihood of false positives.

[3] There are complicated rules about what happens if you violate this guideline (including some parts which are left implementation-defined), but they are largely irrelevant because you should just follow the guideline already.

Raymond Chen

Follow