Another use for the instructions used in atomics

In the section on **atomics** we saw how the ARM V8 load linked / store conditional instructions can be used to create atomic operations on variables in memory.

Here, for review, we present an atomic increment:

```
        .text                                                   // 1
        .p2align    2                                           // 2
                                                                // 3
#if defined(__APPLE__)                                          // 4
        .global     _LoadLinkedStoreConditional                 // 5
_LoadLinkedStoreConditional:                                    // 6
#else                                                           // 7
        .global     LoadLinkedStoreConditional                  // 8
LoadLinkedStoreConditional:                                     // 9
#endif                                                          // 10
1:      ldaxr       w1, [x0]                                    // 11
        add         w1, w1, 1                                   // 12
        stlxr       w2, w1, [x0]                                // 13
        cbnz        w2, 1b                                      // 14
        ret                                                     // 15
```

The nonsense between lines 4 and 10 declare the label in ways compatible with both Apple M and Linux.

The interesting part happens from line 11 through line 14. Line 11 dereferences a pointer to an `int32_t` putting its current value into `w1`. Line 12 is the increment.

Notice the dereference instruction is not the usual `ldr`. Instead it is `ldaxr` which is a dereference that marks the memory location in `x0` as a load for which we're hoping for exclusivity. Hoping.

We don't actually know if we had exclusive access to the memory location until the `stlxr` returns 0, meaning no one else has attempted to change the value at the location.

If `stlxr` doesn't return 0, then the value WE have is stale. So, we try again.

Making a spin-lock

When one has a shared resource used by more than one thread it must be protected. This is the nugget to be aware of when working with threads.

Take a look at this thread worker:

```
void Worker(int32_t id) {                                       // 1
    int32_t counter = 0;                                        // 2
    while (counter < 4) {                                       // 3
        Lock(&lock_variable);                                   // 4
```

```
        counter++;                                                      // 5
        cout << "thread: " << id << " counter: " << counter << endl;// 6
        std::this_thread::sleep_for(chrono::milliseconds(5));       // 7
        Unlock(&lock_variable);                                         // 8
        sched_yield();                                                  // 9
    }                                                                   // 10
}
```

The purpose of the worker is to print something to the console 4 times then exit. The shared resource is the console itself. Without protecting the console, threads will step over each other trying to print to it.

Here is a sample of what could happen without our spin-lock:

```
thread: 0thread: 3 counter: 1
thread: 7 counter: 1 counter: thread:
thread: thread: 10thread: 5 counter: 1
thread:  counter: thread: 121 counter:
thread: 8 counter: 113
thread: thread: 2thread:  counter: 151 counter:
```

With our spin-lock, here's what we might get:

```
thread: 12 counter: 3
thread: 4 counter: 2
thread: 7 counter: 4
thread: 3 counter: 2
thread: 1 counter: 4
thread: 2 counter: 4
thread: 13 counter: 3
thread: 12 counter: 4
```

Line 7 stresses the lock.

Line 9 causes the currently running thread to voluntarily deschedule. This makes the output more interesting. With out it, after unlocking, the same thread may regain the lock immediately.

Now let's look at the spin-lock. But first, a spin-lock is called a spin-lock because a thread that doesn't get the lock will `spin` trying to get it. This wastes time and generates heat, using electricity. Bummer.

Here is the source code to the spin-lock for ARM V8.

```
#if defined(__APPLE__)                                          // 1
_Lock:                                                          // 2
#else                                                           // 3
Lock:                                                           // 4
#endif                                                          // 5
        START_PROC                                              // 6
1:      ldaxr       w1, [x0]                                    // 7
```

```
        cbnz        w1, 1b          // lock taken - spin.          // 8
        add         w1, w1, 1                                      // 9
        stlxr       w2, w1, [x0]                                   // 10
        cbnz        w2, 1b          // shucks - somebody meddled. // 11
        // considered using dmb here                               // 12
        ret                                                        // 13
        END_PROC                                                   // 14
```

Once again, line 7 does a `ldaxr` dereferencing the lock itself (once again an `int32_t`) and marks the location of the lock as being hopefully, exclusive.

Having gotten the value of the lock, on line 8, its value is inspected and if found to be non-zero, we branch back to attempting to get it again - this is the spin.

If the contents of the lock is 0, its value in `w1` is changed to non-zero. Note, this could be made a bit better if a value of 1 was stored in another `w` register and simply used directly on line 10.

Line 10 conditionally stores the changed value back to the location of the lock. If the `stlxr` returns 0, we got the lock. If not, we start over - somebody else got in there ahead of us. Perhaps this happened because we were descheduled. Perhaps we lost the lock to another thread running on a different core.

The unlock looks like this:

```
#if defined(__APPLE__)                                             // 1
_Unlock:                                                           // 2
#else                                                              // 3
Unlock:                                                            // 4
#endif                                                             // 5
        START_PROC                                                 // 6
        str         wzr, [x0]                                      // 7
        // considered using dmb here                               // 8
        ret                                                        // 9
        END_PROC                                                   // 10
```

All it does is set to value of the lock to zero. The correct operation of the lock requires that no bad actor simply stomps on the lock by calling `Unlock` without first owning the lock. Just say no to lock stompers.

Please see the source code located here for some additional comments regarding the implementation.