A word on C11/C++11 low-level atomics

```
std::atomic<int> flag0(0),flag1(0),turn(0);

void lock(unsigned index) {
    if (0 == index) {
        flag0.store(1, std::memory_order_relaxed);
        turn.exchange(1, std::memory_order_acq_rel);

        while (flag1.load(std::memory_order_acquire)
            && 1 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    } else {
        flag1.store(1, std::memory_order_relaxed);
        turn.exchange(0, std::memory_order_acq_rel);

        while (flag0.load(std::memory_order_acquire)
            && 0 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    }
}

void unlock(unsigned index) {
    if (0 == index) {
        flag0.store(0, std::memory_order_release);
    } else {
        flag1.store(0, std::memory_order_release);
    }
}
```

Atomic variable declaration

New syntax for memory accesses

Qualifier

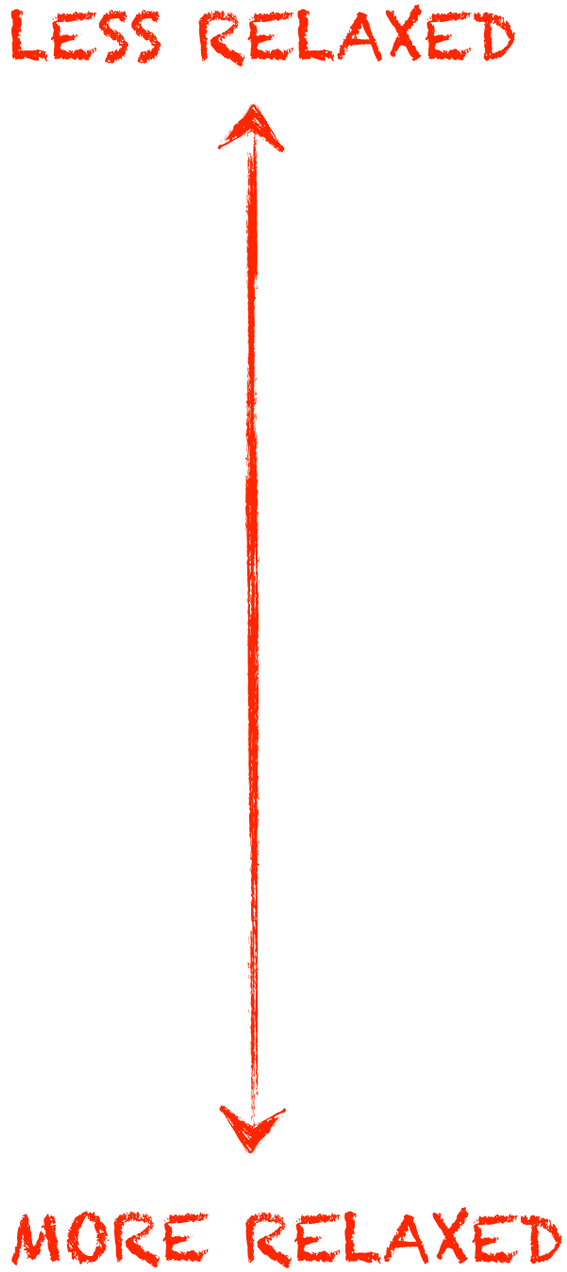
Low-level atomics

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED



MO_SEQ_CST

The compiler must ensure that **`MO_SEQ_CST`** accesses have sequentially consistent semantics.

Thread 0	Thread 1
`x.store(1,MO_SEQ_CST)`	`y.store(1,MO_SEQ_CST)`
`r1 = y.load(MO_SEQ_CST)`	`r2 = x.load(MO_SEQ_CST)`

The program above cannot end with `r1 = r2 = 0`.

Sample compilation on x86:

store: **`MOV; MFENCE`**
load: **`MOV`**

Sample compilation on Power:

store: **`HWSYNC; ST`**
load: **`HWSYNC; LD; CMP; BC; ISYNC`**

MO_RELEASE / MO_ACQUIRE

Supports a fast implementation of the message passing idiom:

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`r1 = y.load(MO_ACQUIRE)`
`y.store(1,MO_RELEASE)`	`r2 = x.load(MO_RELAXED)`

The program above cannot end with `r1 = 1` and `r2 = 0`.

Accesses to the data structure can be reordered/optimised (`MO_RELAXED`).

Sample compilation on x86:

store: `MOV`
load: `MOV`

Sample compilation on Power:

store: `LWSYNC; ST`
load: `LD; CMP; BC; ISYNC`

MO_RELEASE / MO_CONSUME

Supports a fast implementation of the message passing idiom on Power:

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`r1 = y.load(x,MO_CONSUME)`
`y.store(&x,MO_RELEASE)`	`r2 = (*r1).load(MO_RELAXED)`

The program above cannot end with `r1 = 1` and `r2 = 0`.

The two loads have an address dependency, Power won't reorder them.

Sample compilation on x86:

store: `MOV`
load: `MOV`

Sample compilation on Power:

store: `LWSYNC; ST`
load: `LD`