MANCHESTER 1824
The University of Manchester

COMP35112 Chip Multiprocessors

Load-Linked and Store-Conditional

Pierre Olivier

Load-Linked and Store-Conditional

- LL/SC used in RISC processors: ARM, IBM Power, etc.
- 2 instructions:
  - **Load-linked**
  - **Store-conditional**

Load-Linked and Store-Conditional

- LL/SC used in RISC processors: ARM, IBM Power, etc.
- 2 instructions:
  - **Load-linked**
  - **Store-conditional**
- Slightly different from traditional loads and stores
  - Additional effects on CPU state

Load-Linked and Store-Conditional

- LL/SC used in RISC processors: ARM, IBM Power, etc.
- 2 instructions:
  - **Load-linked**
  - **Store-conditional**
- Slightly different from traditional loads and stores
  - Additional effects on CPU state
- **Can act as a pair atomically without holding the bus until completion**

Load-Linked

```
ldl %r1, %r2
```

- `%r1` ← `*(%r2)`
- In addition, **core keeps some state for the `ldl`**:
  - Sets *load linked flag*
  - Saves the address (from `%r2`) in the *locked address register*

Store-Conditional

```
stc %r1, %r2
```

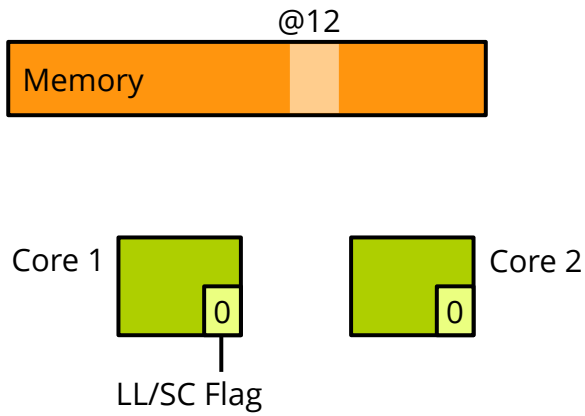
- *Tries* to do `*(%r2)` ← `%r1`
  - **Only succeeds if load linked flag set**
- Afterwards value of load linked flag returned in `%r1`
- And the load link flag is cleared

The Load-Linked Flag

- **Load linked flag is cleared if another core writes to the locked address**

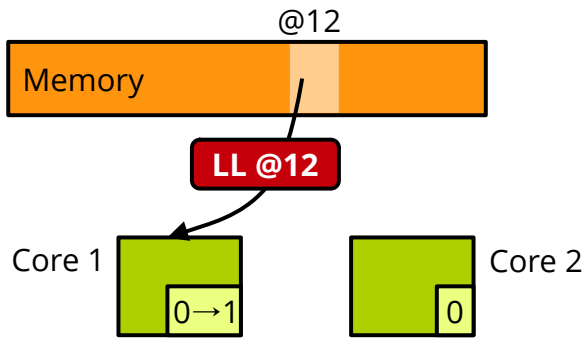
The Load-Linked Flag

- **Load linked flag is cleared if another core writes to the locked address**



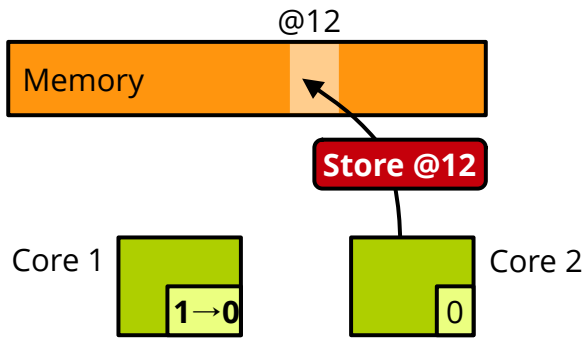
The Load-Linked Flag

- **Load linked flag is cleared if another core writes to the locked address**



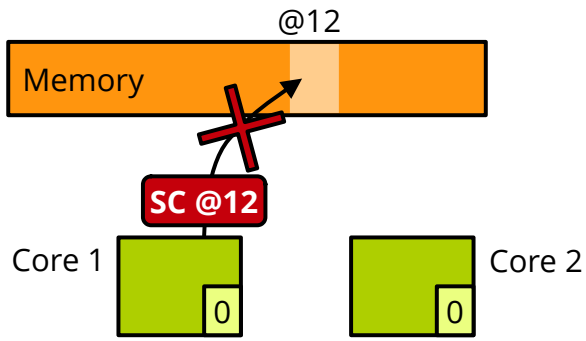
The Load-Linked Flag

- **Load linked flag is cleared if another core writes to the locked address**



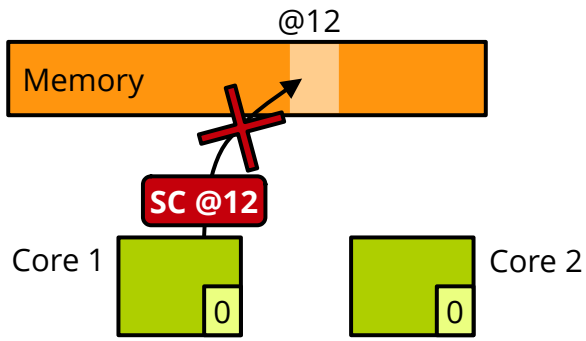
The Load-Linked Flag

- **Load linked flag is cleared if another core writes to the locked address**



The Load-Linked Flag

- **Load linked flag is cleared if another core writes to the locked address**



- Other events clearing the flag: context switches, interruptions
- **Allows to be sure that a `ll`/`sc` pair executed atomically with respect to the locked address**

Our Semaphore with LL/SC

```
/* Address of the lock in %r2 */

loop: ldl %r1, %r2
      comp $0, %r1  /* S == 0 (semaphore already taken)? */
      beq loop      /* if so, try again */
      mov $0, %r1   /* looks like it's free, prepare to take the semaphore */
      stc %r1, %r2  /* Try to take it */
      cmp $1, %r1   /* Did the write succeed? */
      bne loop      /* If not, someone beat us to it... try again */

      /* critical section here... */

      st $1, %r2    /* release the semaphore with a simple store */
```

Our Semaphore with LL/SC

```
/* Address of the lock in %r2 */

loop: ldl %r1, %r2
      comp $0, %r1  /* S == 0 (semaphore already taken)? */
      beq loop      /* if so, try again */
      mov $0, %r1   /* looks like it's free, prepare to take the semaphore */
      stc %r1, %r2  /* Try to take it */
      cmp $1, %r1   /* Did the write succeed? */
      bne loop      /* If not, someone beat us to it... try again */

      /* critical section here... */

      st $1, %r2    /* release the semaphore with a simple store */
```

- Highlighted code executes atomically with respect to the memory location pointed by `%r2`
  - Any write to that location/interrupt/context switch during the atomic part will lead to `stc` failing

The Power of LL/SC

- Single atomic RMW instructions such as `tas` atomically combine load (R) and store (W) operations, and a potential in-register update (M)

The Power of LL/SC

- Single atomic RMW instructions such as `tas` atomically combine load (R) and store (W) operations, and a potential in-register update (M)
- The code starting with LL and ending with SC is *not atomic in absolute*
  - Even for a LL directly followed by a LC

The Power of LL/SC

- Single atomic RMW instructions such as `tas` atomically combine load (R) and store (W) operations, and a potential in-register update (M)
- The code starting with LL and ending with SC is *not atomic in absolute*
  - Even for a LL directly followed by a LC
  - However LL/SC allows to determine **if anything between the LL and the SC has executed atomically or not, with respect to the corresponding address**

Spinlock

- All our implementations of `wait(S)` use a busy loop
  - Called **busy waiting** or **spinning**
  - **Spinlocks**

Spinlock

- All our implementations of `wait(S)` use a busy loop
  - Called **busy waiting** or **spinning**
  - **Spinlocks**
- Hurt performance when contended

Spinlock

- All our implementations of `wait(S)` use a busy loop
  - Called **busy waiting** or **spinning**
  - **Spinlocks**
- Hurt performance when contended
- **Sleeping would be more efficient**
  - Such higher-level locks can be called *mutexes*
  - Requires OS support

Summary

- Atomic RMW instructions: inefficient/hard to implement in some situations
- LL/SC address these problems
  - Atomic execution with respect to a particular memory location
  - Without locking the cache coherence bus
- Next lecture: can we use such atomic instructions, **instead of locks**?