## Synchronization

Critical (atomic) section access -> thread synchronization (only one thread can in critical section)

## Mutual Exclusion

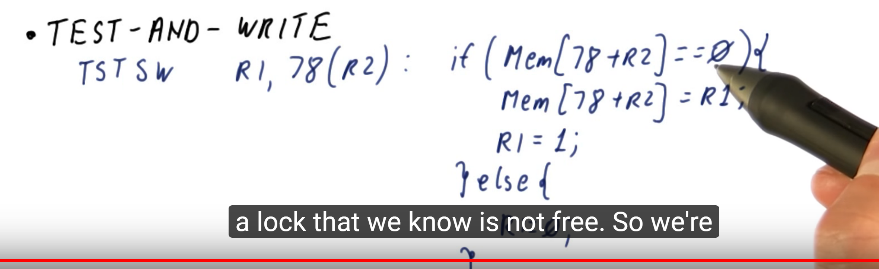
1. Lock -> preventing critical section interleaving

Use special atomic operation section to check and write lock variable

* Atomic Instructions
  1. Atomic exchange (do both load and store, swap content)



* Test-and-Write
  1. If satisfy condition, then write



图片包含 人员, 文字

描述已自动生成

Atomic Exchange will keep try to write to the lock; Test-And-Write will only test the lock value and only write to it when lock is release. (Good, consider coherence)

If keep write, the lock copy will be invalidated and generate bus traffic. For Test-And-Write, since most are read, each will share the lock variable. And the unlock will invalidate the value and everybody sees the new value and try to grab the new locks.

But above instructions are weird, not like store nor load -> Load Linked/ Store Conditional (LL/ SC), separate the read and write, but make them looks like an atomic instruction

图片包含 物体

描述已自动生成

When snooping write to Lock, set the Link register (R1) as 0. Then If the lock is already written by others, the Link register will not be the same as R1, then SC will fail. Rely on coherence.

Example

图片包含 文字, 白板

描述已自动生成

图片包含 文字

描述已自动生成

LL will load variable from Lockvar and store in R2. SC will store the variable in R1 to Lockvar. Initially, R1 will be 1. Link register is hidden register and should only be check through R1 and R2, but not directly read its value.

* First check: check whether R2 is 0. (0 means load successfully)
* Second Check: check whether R1 is stored successfully to Lockvar (R1=1 means store successfully, 0 for failure).

## Coherence and Lock

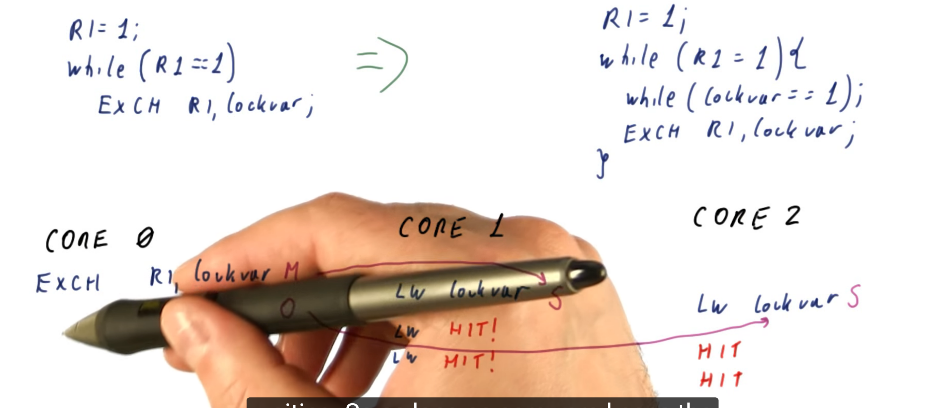
There is lockvar transfer when core trying to grab lock, since it is a write instruction. And the modify->invalid will happen many times until lockvar is released by the holder. There are lots of energy used in data transfer and cache miss will slow the performance.

图片包含 文字

描述已自动生成

Test-and-Atomic-Op Lock

* Busy wait uses normal read, instead of keeping writing to Lockvar. And there is no cache miss, since there is only one write Lockvar and it is in share state.



If somebody write the Lockvar, it will be invalid in current core and generate write miss. (Coherence)

图片包含 文字

描述已自动生成

## Barrier Synchronization

Ensure the task (parallel manner) in each core is finished. Check whether it is the last to arrive. If not, just spin and wait for other cores to arrive.

Flip wanted variable. The reusable barrier then handles the difference in reading value of release.

图片包含 文字

描述已自动生成