



信息科学与技术学院

School of Information Science and Technology

CS 110

Computer Architecture

Thread-Level Parallelism II

Instructors:

Chundong Wang, Siting Liu & Yuan Xiao

Course website: <https://toast->

[lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html](https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html)

School of Information Science and Technology (SIST)

ShanghaiTech University

2025/5/8

Administratives

- Mid-term II tentatively May 15th 8am-10am; you can bring 2-page A4-sized double-sided cheat sheet, **handwritten** only! (**Teaching center 201/301/303**); From start to May 13th lecture (Thread-level parallelism).
- Project 2.2 released, ddl May 19th.
- Project 3 to be released soon. Speed Competition! ddl May 29th
- HW 6 released, ddl approaching May 12th!
- Lab 12 released. **Prepare in advance!**
 - To check May 13th, 15th & 19th;
 - May 12th Monday lab session to check Lab 11;
- Discussion May 9th & 12th on SIMD & OpenMP.

Parallelism Overview

- **Parallel Requests**
Assigned to computer
e.g., Search “CS110”
- **Parallel Threads**
Assigned to core
e.g., Lookup, Ads
- **Parallel Instructions**
>1 instruction @ one time
e.g., 5 pipelined instructions
- **Parallel Data**
>1 data item @ one time
e.g., Add of 4 pairs of words
- **Hardware descriptions**
All gates @ one time
- **Programming Languages**

Software

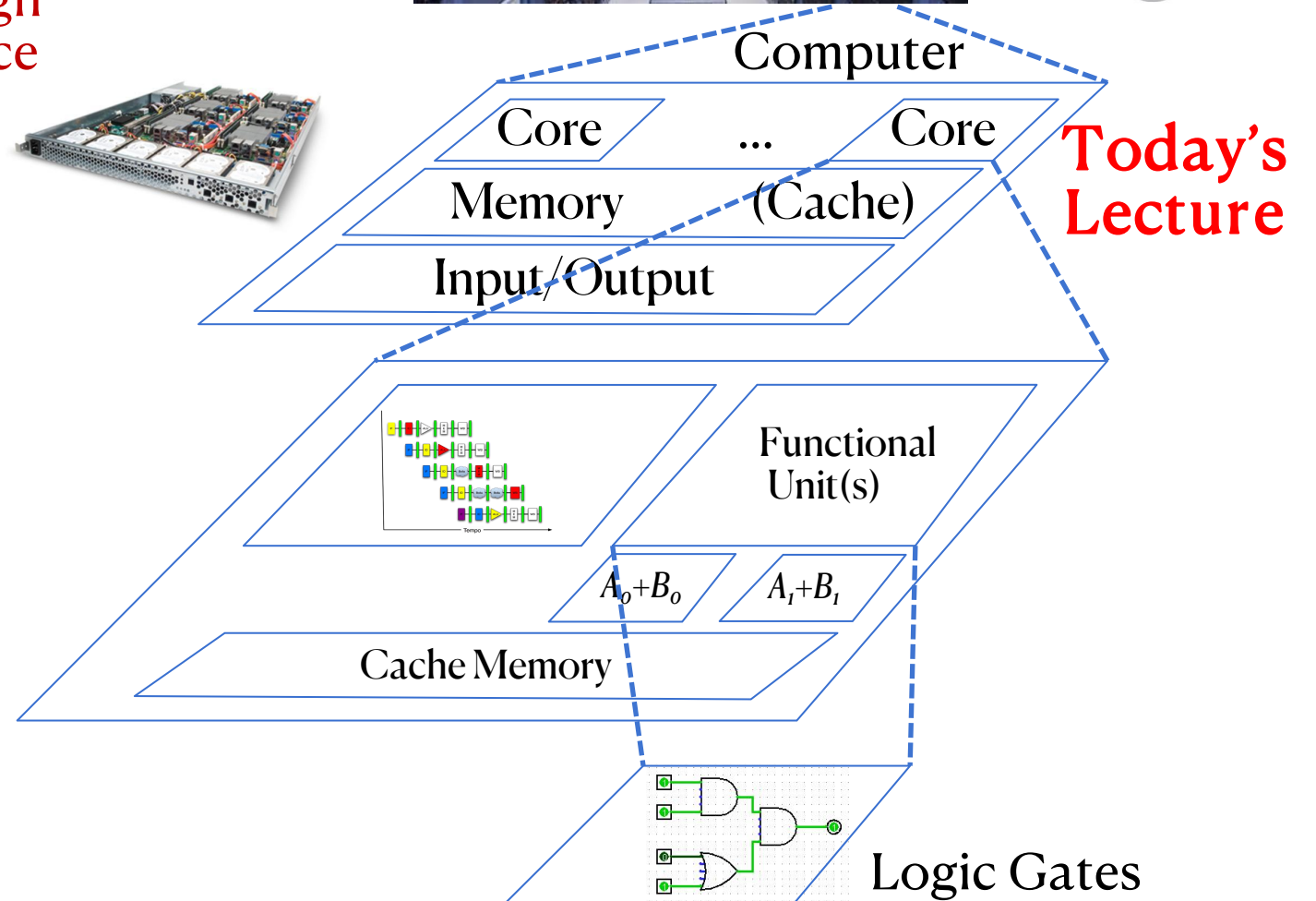
Hardware

**Harness
Parallelism &
Achieve High
Performance**

Warehouse
Scale
Computer



Smart
Phone



Synchronization

- To enforce multithreaded program correctness, we often need to **synchronize** threads, i.e., coordinate their execution.
 - Most commonly, know when one task is finished writing so that it is safe for another to read.
- Desired correct outcome:

```
load
addi
store
```

```
load
addi
store
```

```
load
addi
store
```

```
load
addi
store
```

(or any permutation between these four code segments)

- Note: If we enforce the above, then execution effectively becomes sequential...(Amdahl's Law)

Critical Sections with OpenMP (1/2)

- A **critical section** is a segment of code that must be executed by a single thread at a time, thereby enforcing **synchronization**.
 - In OpenMP, you can declare critical sections of code.
 - Each thread can safely execute code in critical section, knowing that it is the only thread that can execute that section at that time
 - This user-level specification (e.g., in OpenMP) relies on hardware synchronization instructions (e.g., in RISC-V) (more later)
- `#pragma omp barrier` [\[docs\]](#)
 - Forces all threads to wait until all threads have hit the barrier
- `#pragma omp critical` [\[docs\]](#)
 - Creates a critical section within a parallel code segment; only one thread can run a critical section at a time.

OpenMP Hello World, Synchronized

```
#include <stdio.h>
#include <omp.h>
int main() {
    int x = 0;                                /* shared variable */
    #pragma omp parallel
    {
        int tid = omp_get_thread_num(); /* private variable */
        #pragma omp critical
        {
            x++;
            printf("Hello World from thread = %d, x = %d\n", tid, x);
        }
        #pragma omp barrier
        if (tid == 0) {
            printf("Number of threads = %d\n", omp_get_num_threads());
        }
    }
}
```

Critical Sections with OpenMP (2/2)

- A **critical section** is a segment of code that must be executed by a single thread at a time.
 - In OpenMP, you can declare critical sections of code.
 - Compile to atomic instructions that enable synchronization between threads (more later, RISC-V)
- OpenMP has very restrictive parallelism.
 - Really only good for parallelizing loops...
 - Beyond that:
 - If your critical section is too large, then effectively serial program
→ **Amdahl's law** quickly rears its ugly head
 - If critical sections not defined well, can run into **deadlock**.

The Other Parallel Programming Languages

ActorScript	Concurrent Pascal	JoCaml	Orc
Ada	Concurrent ML	Join	Oz
Afnix	Concurrent Haskell	Java	Pict
Alef	Curry	Joule	Reia
Alice	CUDA	Joyce	SALSA
APL	E	LabVIEW	Scala
Axum	Eiffel	Limbo	SISAL
Chapel	Erlang	Linda	SR
Cilk	Fortran 90	MultiLisp	Stackless Python
Clean	Go	Modula-3	SuperPascal
Clojure	Io	Occam	VHDL
Concurrent C	Janus	occam- π	XC

Mostly
dead

The Other Parallel Programming Languages

- Library – e.g.:

- pthread

- C++:

- std::thread C++11

- std::jthread C++20

- std::mutex; std::lock_guard; std::scoped_lock; std::shared_lock

- std::condition_variable; std::counting_semaphore; std::latch; std::barrier

- std::promise; std::future

- Qt QThread

- OpenMP

Data Race & Synchronization

- Two memory accesses form a **data race** if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get deterministic behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions
- (more later)

Analogy: Buying Milk

- Lilei and roommate Lihua would like to buy a carton of milk
 - Originally no milk;
 - Shared fridge; intend to be exactly one carton in the fridge;

```
// attempt 1  
if milk not in fridge:  
    buy milk at store  
    put milk in fridge
```

- What if Lilei get home while Lihua is out buying milk?
 - Result: Two milk cartons!



Analogy: Buying Milk

- Lilei and roommate Lihua would like to buy a carton of milk
 - Originally no milk;
 - Shared fridge; intend to be exactly one carton in the fridge;

```
// attempt 2, with note
if note not on fridge:
    if milk not in fridge:
        put note on fridge;
        buy milk at store;
        put milk in fridge;
        take note off fridge;
    else no action;
```

Seems good, but ...



Analogy: Buying Milk

- Even with shared note, we can run into a two-milk situation

// Lilei's thread

if note not on fridge:
 if milk not in fridge:

 put note on fridge;

 buy milk at store;
 put milk in fridge;
 take note off fridge;

// Lihua's thread

if note not on fridge:
 if milk not in fridge:

 put note on fridge;

 buy milk at store;
 put milk in fridge;
 take note off fridge;

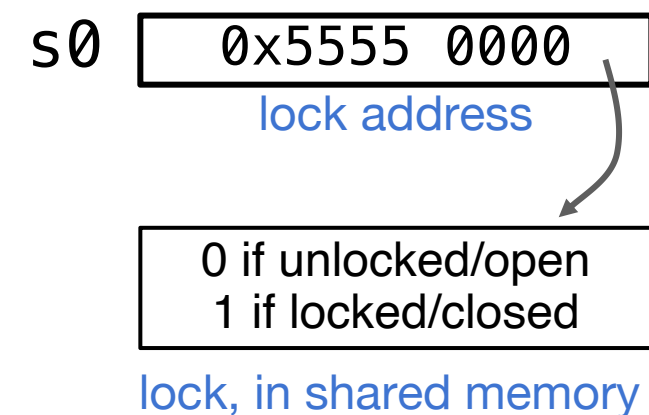
time



Even when threads execute in parallel, they still sequentially access shared resources.

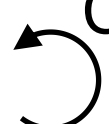
Lock Synchronization

- Use a “Lock” to grant access to a region (**critical section**) so that only one thread can have the lock and operate at a time
 - Need all processors to be able to access the lock, so use a location in shared memory as **the lock**
- Processors read lock and either wait (if locked) or set lock and go into critical section
 - 0 means lock is free / open / unlocked / lock off
 - 1 means lock is set / closed / locked / lock on
- Locks are one approach to implementing thread synchronization.
- Suppose
 - Lock in shared memory @ 0x5555 0000



Lock Synchronization

- Use a “Lock” to grant access to a region (**critical section**) so that only one thread can have the lock and operate at a time
 - Need all processors to be able to access the lock, so use a location in shared memory as **the lock**
- Processors read lock and either wait (if locked) or set lock and go into critical section
 - 0 means lock is free / open / unlocked / lock off
 - 1 means lock is set / closed / locked / lock on
- Pseudocode:

Check lock  Can loop/idle here if locked
Set the lock
Critical section
(e.g. change shared variables)
Unset the lock

Lock Synchronization


- Use a “Lock” to grant access to a region (**critical section**) so that only one thread can have the lock and operate at a time
 - Need all processors to be able to access the lock, so use a location in shared memory as **the lock**
- Two operations, formally:
 - Acquire: try to acquire the lock. If successful, keep going. Otherwise, wait and try later;
 - Release: Unlock and continue (only works if we originally had the lock)

```
// attempt 3, with lock
acquire fridgelock
  if milk not in fridge:
    buy milk at store
    put milk in fridge
  release fridgelock
```


Lock Synchronization

- When Lihua does not have the lock, then **wait**

```
// Lilei  
acquire fridgelock  
  
    if milk not in fridge:  
        buy milk at store  
        put milk in fridge  
release fridgelock
```

```
// Lihua  
acquire fridgelock  Can loop/idle here  
                    if locked (wait)  
  
    if milk not in fridge:  
        buy milk at store  
        put milk in fridge  
release fridgelock
```

Locks inherently enforce some serialization of threads. Amdahl's Law strikes again!

Possible Naive Lock Implementation

- Lock (a.k.a. busy wait) in RISC-V (acquire)

```
Get_lock:                # s0 -> addr of lock
    addi    t1,zero,1      # t1 = Locked value
Loop:    lw    t0,0(s0)     # load lock
        bne    t0,zero,Loop # loop if locked
Lock:    sw    t1,0(s0)     # Unlocked, so lock
```

- Unlock (release)

```
Unlock:
    sw    zero,0(s0)
```

- Any problems with this?

Naive Lock Problem

- Thread 1 acquire

```
addi t1,zero,1
```

```
Loop: lw t0,0(s0)
```

```
bne t0,zero,Loop
```

```
Lock: sw t1,0(s0)
```

- Thread 2 acquire

```
addi t1,zero,1
```

```
Loop: lw t0,0(s0)
```

```
bne t0,zero,Loop
```

```
Lock: sw t1,0(s0)
```

Time

**Both threads think they have set the lock!
Exclusive access not guaranteed!**

Actually it resembles the “note”



Hardware Synchronization

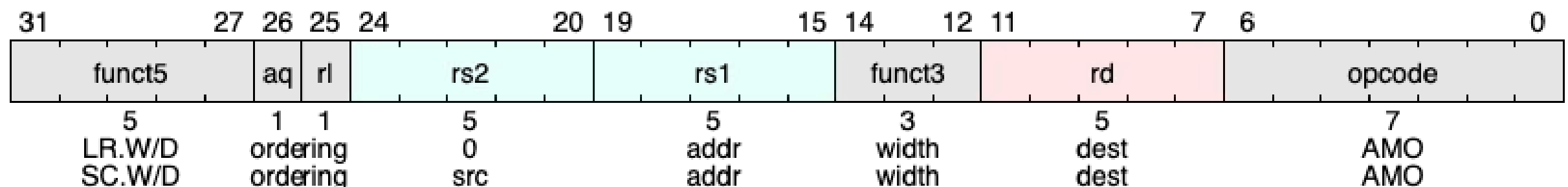
- Hardware support required to prevent an interloper (another thread) from changing the value
 - **Atomic** read/write memory operation (all other operations must to happen strictly before/after the read/write)
 - No other access to the location allowed between the read and write
- How to implement in software?
 - Single instruction: atomic swap of register \leftrightarrow memory through atomic instructions;
 - Pair of instructions: one for read (and lock), one for write (and unlock);
- Needed even on uniprocessor systems
 - Interrupts can happen: can trigger thread context switches...

RISC-V: Two solutions!

- Option 1: Read/Write Pairs
 - Pair of instructions for “linked” read and write
 - Load-reserved and Store-conditional
 - No other access permitted between read and write
 - Must use shared memory (multiprocessing)
- Option 2: Atomic Memory Operations
 - Atomic swap of register \leftrightarrow memory

Option 1: Read/Write Pairs

- Load-reserved instruction: `lrr rd, rs`
 - Load the word (doubleword) pointed to by `rs` into `rd`, and register a (hardware thread) reservation set 
- Store-conditional: `sc rd, rs1, rs2`
 - Store the value in `rs2` into the memory location pointed to by `rs1`, only if the reservation is still valid and set the status in `rd`
 - Returns 0 (success) to `rd` if location has not changed since the `lrr`
 - Returns nonzero (failure) to `rd` if location has changed:
Actual store will not take place
 - Invalidate the (hardware thread) reservation whether success or not 



lr/sc Example

- Atomic swap (to test/set lock variable)
- Exchange contents of register and memory: $s4 \leftrightarrow \text{Mem}(s1)$

```

try:
lr      t1, s1      #load reserved
sc      t0, s1, s4  #store conditional
bne     t0, x0, try #loop if sc fails
add     s4, x0, t1  #load value in s4

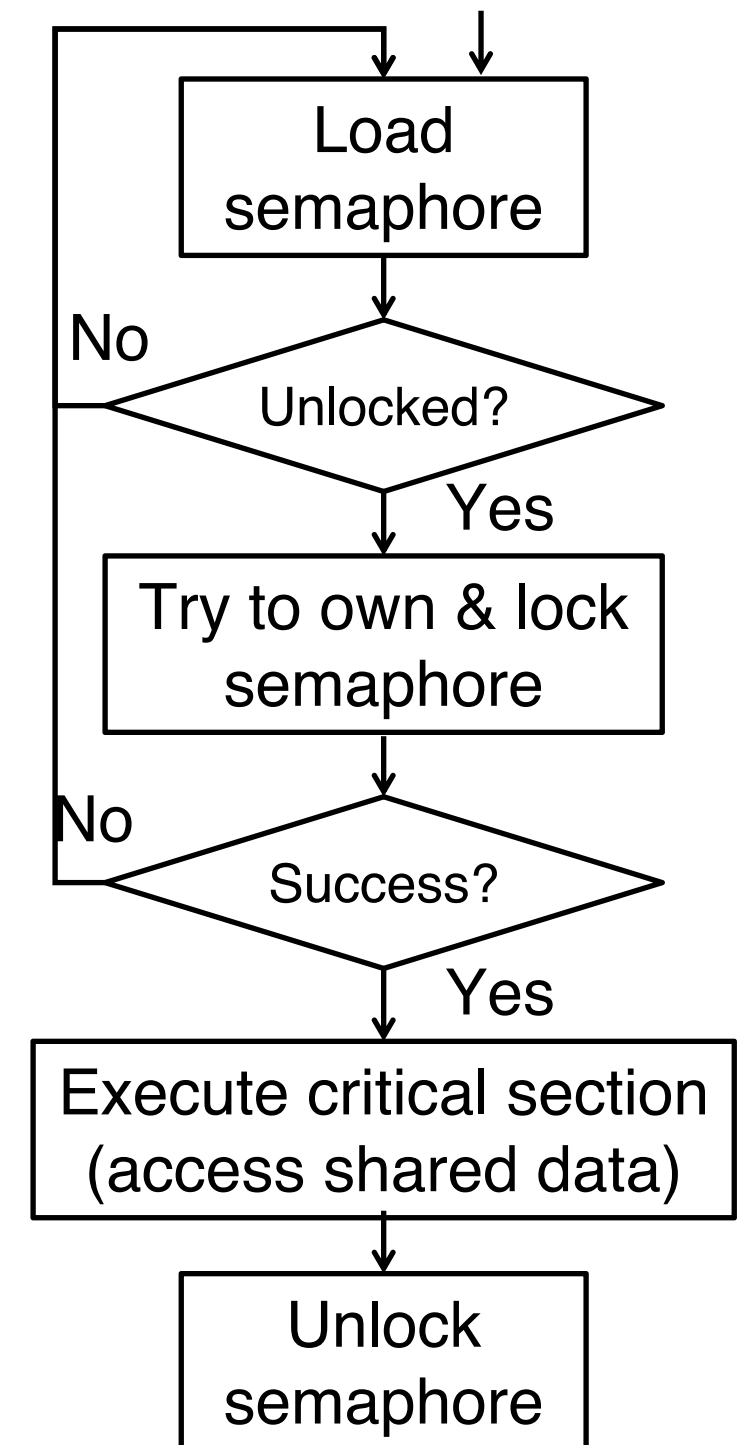
```

sc would fail if another threads
executes **sc** before here

Load-reserved instruction: `lr rd, rs`
Store-conditional: `sc rd, rs1, rs2`

Test-and-Set

- In a single atomic operation:
 - Test to see if a memory location is set (contains a 1)
 - Set it (to 1) if it isn't (it contained a zero when tested)
 - Otherwise indicate that the Set failed, so the program can try again
 - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations



Test-and-Set in RISC-V using `lr/sc`

- Example: RISC-V sequence for implementing a T&S at (s1)

Try:

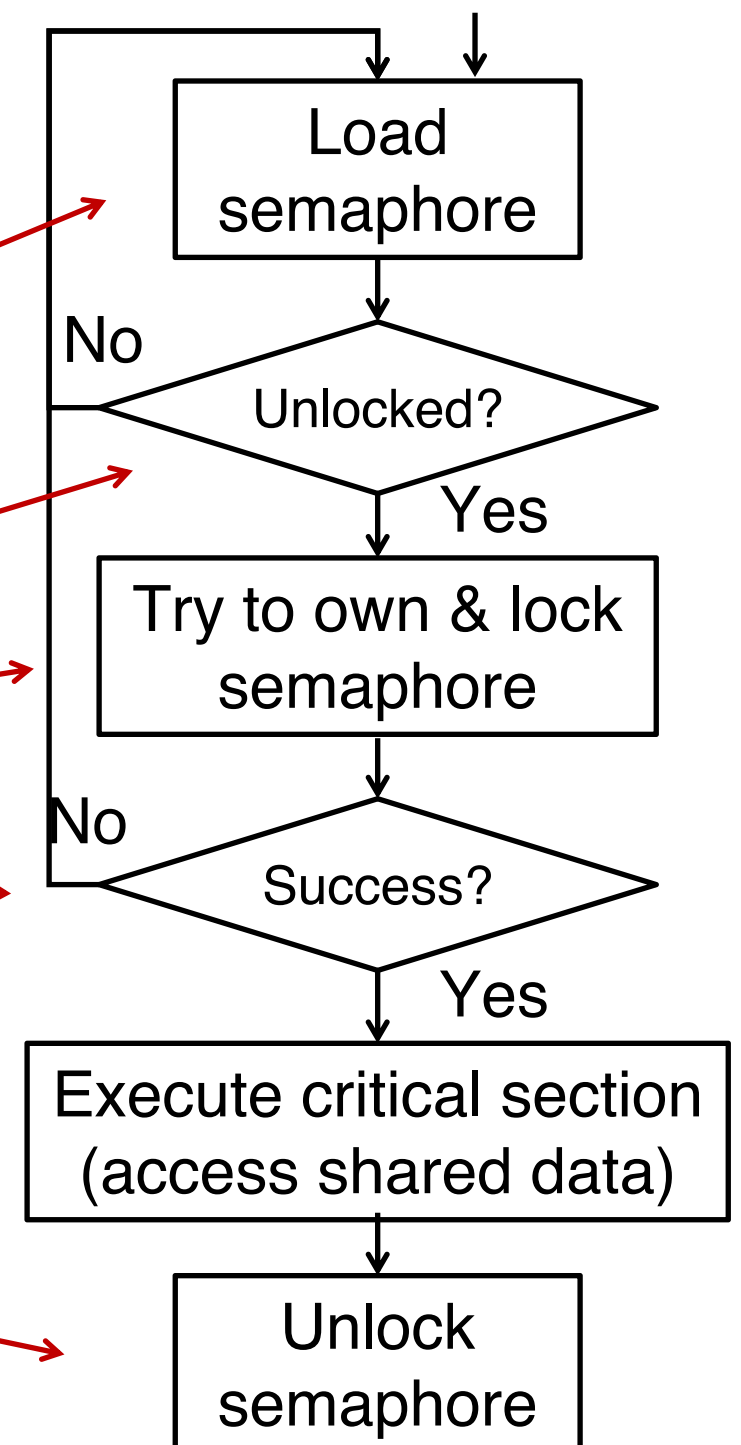
```
li t2, 1
lr t1, s1
bne t1, x0, Try
sc t0, s1, t2
bne t0, x0, Try
```

Locked:

```
# critical section
```

Unlock:

```
sw x0, 0(s1)
```



Option 2: RISC-V Atomic Memory Operations (AMOs)

- Encoded with an R-type instruction format
 - swap, add, and, or, xor, max, min
 - AMOSWAP rd, rs2, (rs1)
 - AMOADD rd, rs2, (rs1)
- Take the value pointed to by rs1
 - Load it into rd
 - Apply the operation to that value with the contents in rs2
 - If rs2==rd, use the old value in rd
 - Store the result back to where rs1 is pointed to
- This allows atomic swap as a primitive
 - It also allows “reduction operations” that are common to be efficiently implemented

AMO Example

- Assume that the lock is in memory location stored in register a0
- The lock is “set” if it is 1; it is “free” if it is 0 (it’s initial value)

```
li          t0, 1
# Get 1 to set lock
Try: amoswap.w.aq      t1, t0, (a0)
# t1 gets old lock value while we set it to 1
bnez       t1, Try
# if it was already 1, another thread has the lock
# so we need to try again
... critical section goes here ...
amoswap.w.rl      x0, x0, (a0)
# store 0 in lock to release
```

Lock Synchronization Implementation

Broken Synchronization

```
while (lock != 0) ;
```

```
lock = 1;
```

```
// critical section
```

```
lock = 0;
```

Fix (lock is at location (a0))

```
        li            t0, 1  
Try:    amoswap.w.aq   t1, t0, (a0)  
        bnez          t1, Try
```

```
Locked:
```

```
# critical section
```

```
Unlock:
```

```
        amoswap.w.rl   x0, x0, (a0)
```

How to Implement

- Don't implement yourself!
- Use according library – e.g.:
 - pthread
 - C++:
 - `std::thread` C++11
 - `std::jthread` C++20
 - `std::mutex`; `std::lock_guard`; `std::scoped_lock`; `std::shared_lock`
 - `std::condition_variable`; `std::counting_semaphore`; `std::latch`; `std::barrier`
 - `std::promise`; `std::future`
 - Qt QThread
 - OpenMP

<https://en.cppreference.com/w/cpp/thread>

Summary

- Sequential software is slow software
 - SIMD and MIMD only path to higher performance
- Multithreading increases utilization
- OpenMP as simple parallel extension to C
 - Threads, Parallel for, private, critical sections, ...
 - \approx C: small so easy to learn, but not very high level and it's easy to get into trouble
- Synchronization & Lock synchronization
 - Can be implemented by atomic operations in RISC-V
 - `lr/sc` pair or AMO instructions