信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
Thread-Level Paralellism III

Instructors: Siting Liu & Yuan Xiao
Course website: https://faculty.sist.shanghaitech.edu.cn/liust/courses/CS110.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2026/5/26

Administratives

- Project 2.2 released, ddl June 4th.
- Project 3 released, ddl June 11th.
- HW6 released, ddl May 28th!
- Lab 13 to be released, to be checked on Jun. 4th/8th/9th.
- Discussion May 29th on SIMD/profiling/etc., useful for project 3.
- Midterm II marks available, regrade ddl: May 25th, 23:59
- Multiple assignments overlapped, start early!

Parallelism Overview

Software

- 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

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Smart Phone

Today’s Lecture

Synchronization

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

  Desired correct outcome:

(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)

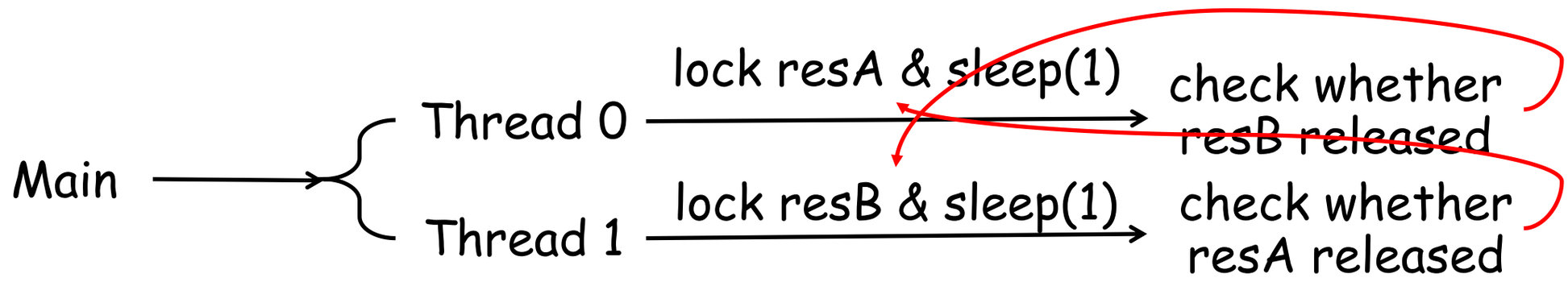
- A critical section is a segment of code that must be executed by a single thread at a time.
  - In OpenMP, you can declare critical sections of code.
  - Compile to atomic instructions that enable synchronization between threads (more later, RISC-V)

- 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.

Deadlock Example

```
int main() {
    int shared_resource_A = 0;
    int shared_resource_B = 0;
    #pragma omp parallel num_threads(2)
    {
        int tid = omp_get_thread_num();
        if (tid == 0) {
            // Thread 0: lock A then B
        #pragma omp critical(resA)
        {   printf("Thread 0 locked A\n");
            sleep(1);  // simulate work
            #pragma omp critical(resB)
            {printf("Thread 0 locked B\n");
                shared_resource_B++;}
        }
}
```

```
else if (tid == 1) {
    // Thread 1: lock B then A  <-- opposite order!
    #pragma omp critical(resB)
    {   printf("Thread 1 locked B\n");
        sleep(1);
        #pragma omp critical(resA)
        {printf("Thread 1 locked A\n");
            shared_resource_A++;}
        }
    }
}
printf("Done\n");
return 0;
}
```



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	Fortan 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 (software) 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

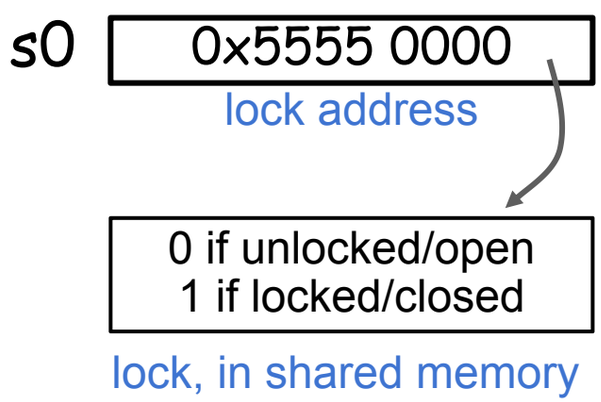
```
// Lilei's thread
if note not on fridge:
    if milk not in fridge:

        put note on fridge;

time

        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;
```

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:

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)

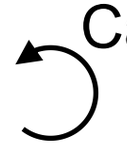
```
// attempt 3, with lock
acquire fridgelock
  if milk not in fridge:
    buy milk at store
    put milk in fridge
release fridgelock
```

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

  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? (**ISA support**)
  - Single instruction: atomic swap of register ↔ memory through atomic instructions;
  - Pair of instructions: one for read (and lock/acquire), one for write (and unlock/release);
- 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 ↔ memory

Option 1: Read/Write Pairs

- Load-reserved instruction: `lr rd,rs` (RV32A, Atomic extension)
  - Load the word (doubleword) pointed to by `rs` into `rd`, and register a (hardware thread) reservation set 🚩

- Store-conditional: `sc rd,rs1,rs2` (RV32A, Atomic extension)

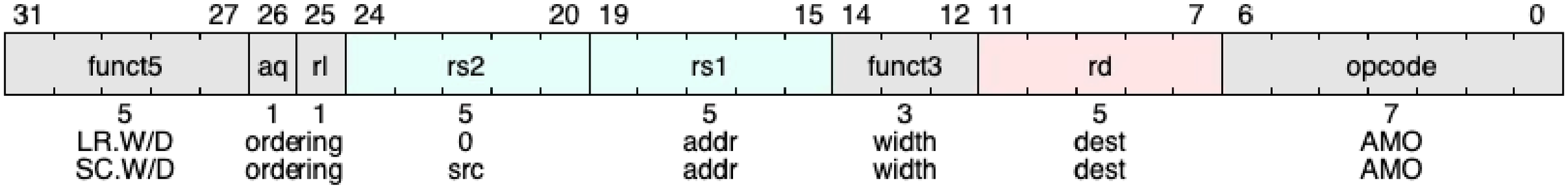
  Store the value in `rs2` into the memory location pointed to by `rs1`, only if the reservation is still valid and set the status in `rd`

  Returns 0 (success) to `rd` if location has not changed since the `lr`

  Returns nonzero (failure) to `rd` if location has changed:

  Actual store will not take place

  Invalid the (hardware thread) reservation whether success or not 🏳



lr/sc Example

- The reservation set itself is not a lock
- Atomic swap (to test/set lock variable)
- Exchange contents of register and memory: s4 ↔ Mem(s1)

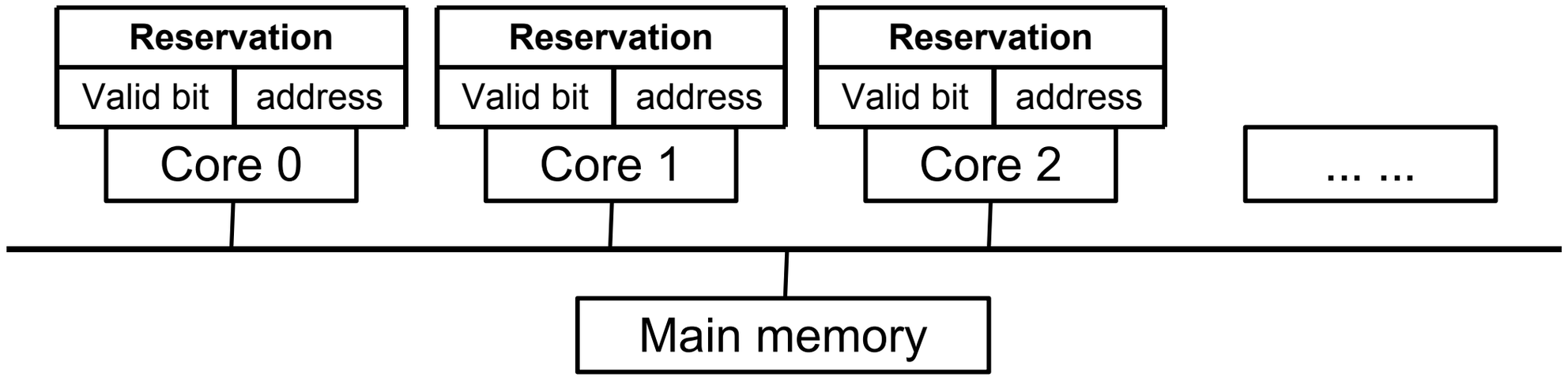
```
try:
lr      t1, s1      #load reserved
sc      t0, s1, s4  #store conditional
bne     t0, x0, try #loop if sc fails
add     s4, x0, t1  #load value in s4
```

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

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

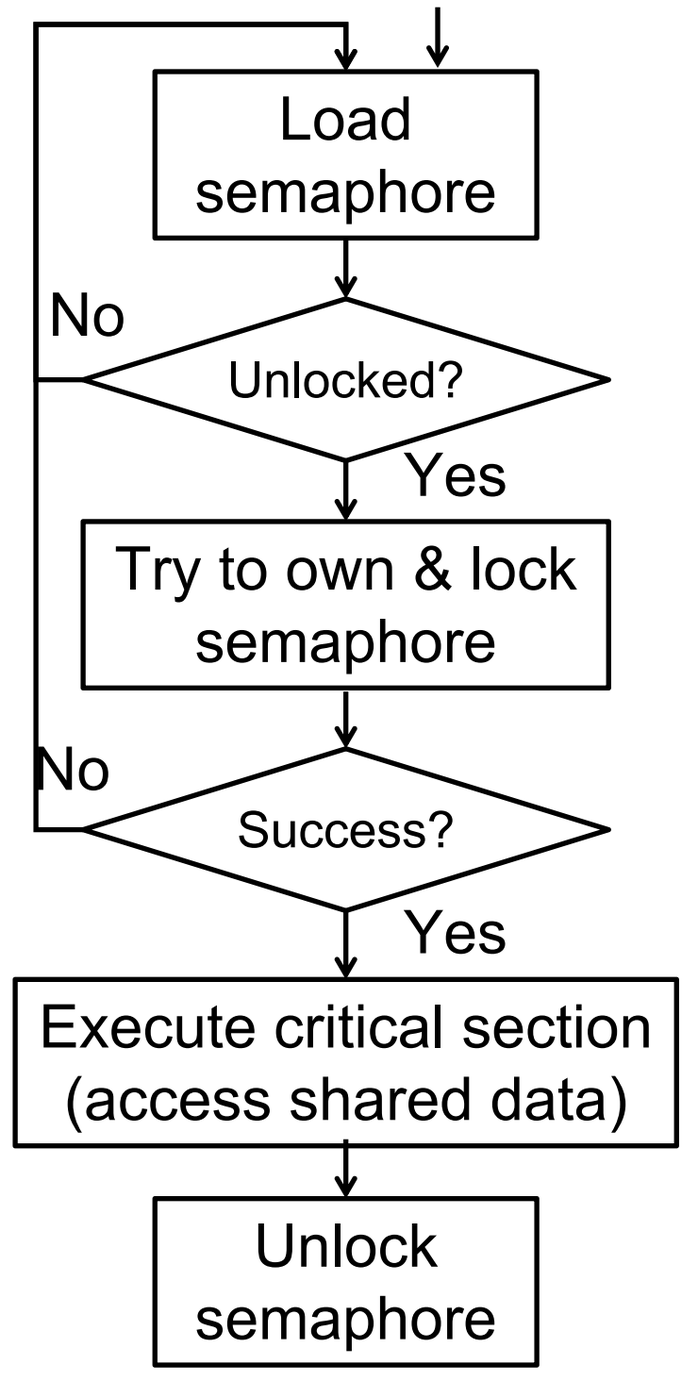
A Possible Naive Implementation

- Load-reserved instruction: `lr rd,rs` (RV32A, Atomic extension)
  - Load the word (doubleword) pointed to by `rs` into `rd`, and register a (hardware thread) reservation set
- Store-conditional: `sc rd,rs1,rs2` (RV32A, Atomic extension)
  - 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 `lr`
    - Returns nonzero (failure) to `rd` if location has changed: Actual store will not take place
  - Invalid the (hardware thread) reservation whether success or not



Test-and-Set

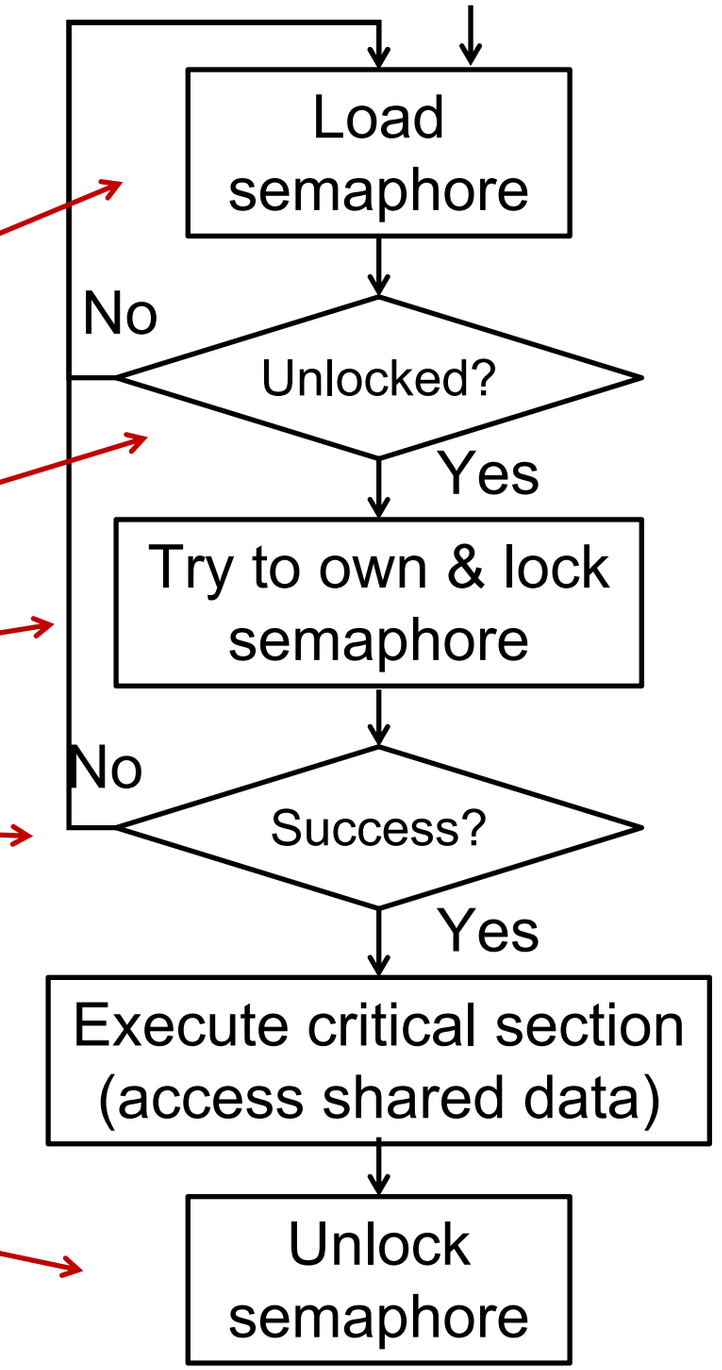
- In a single atomic operation:
  - Test to see if a memory location is set (contains a 1)
  - Set it (to 1) if it isn't (it contained a zero when tested)
    - Otherwise indicate that the Set failed, so the program can try again
- Useful for implementing lock operations
- Avoid ABA situation



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

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

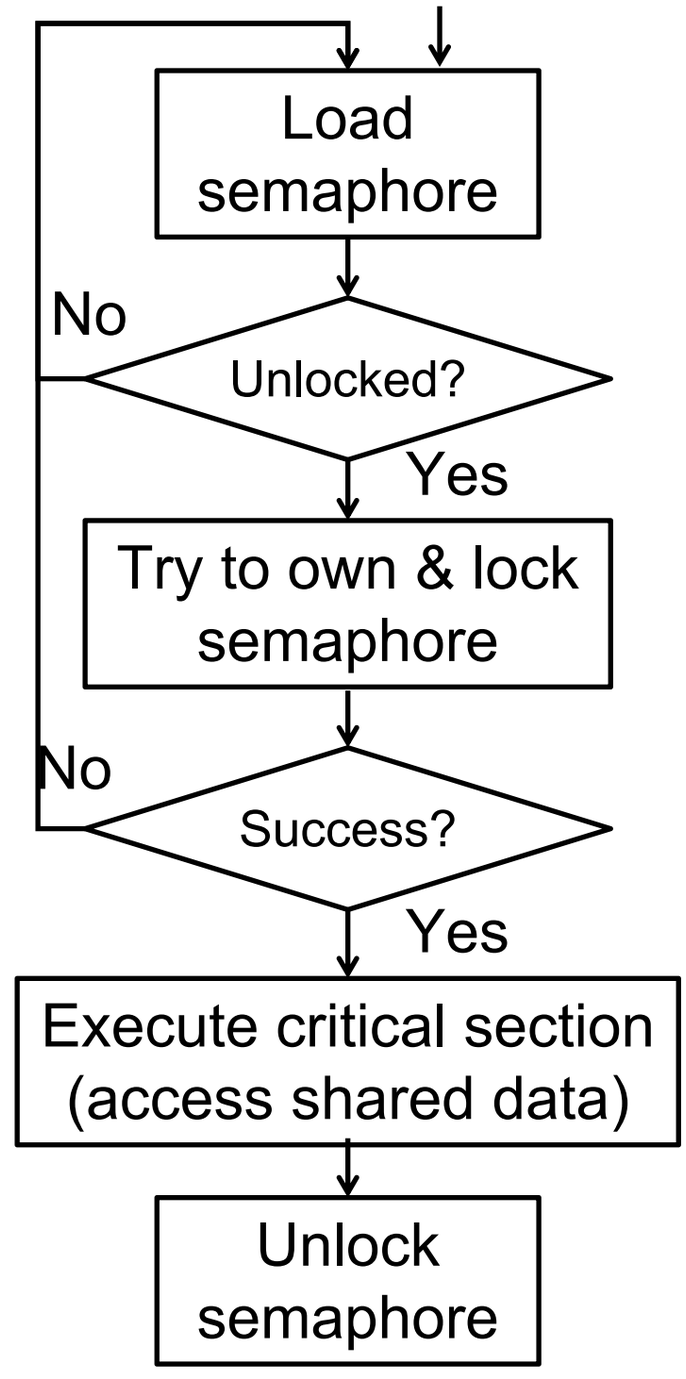
```
        li t2, 1
Try:
        lr t1, s1
        bne t1, x0, Try
        sc t0, s1, t2
        bne t0, x0, Try
Locked:
        # critical section
Unlock:
        sw x0,0(s1)
```



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

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

```
        li t2, 1
Try:
        lr.aqrl t1, (s1)
        bne t1, x0, Try
        sc.aqrl 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)

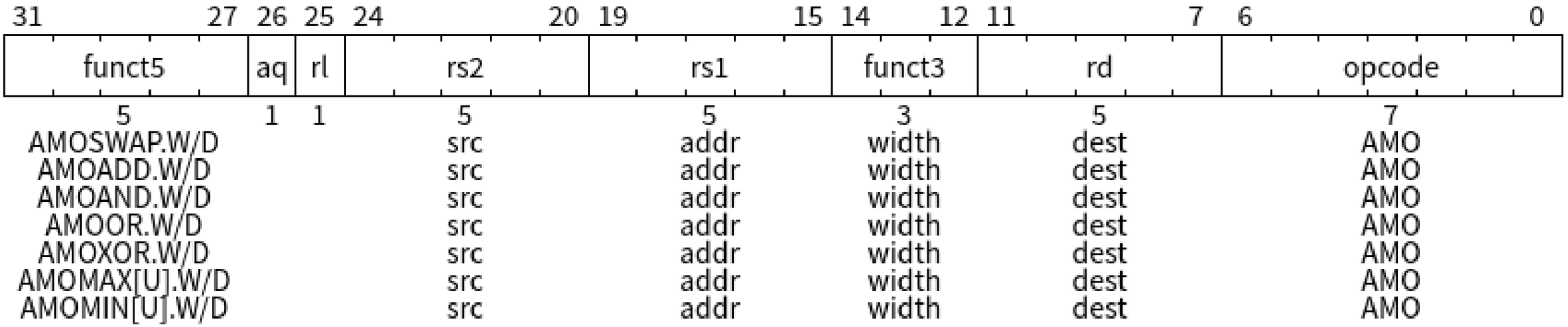
Take the value pointed to by rs1

- 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

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



- Naive implementation in hardware: cache block lock

Tag	VB	LRU	Dirty	**Lock**	DATA			
0x010F	1	2	0					
0x0048	1	1	0					
0x0008	1	3	0					
0x03D0	1	0	0					

https://docs.riscv.org/reference/isa/unpriv/a-st-ext.html hart: hardware thread

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, task/taskwait…
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble
- Synchronization & Lock synchronoization
  - Can be implemented by atomic operations in RISC-V
  - `lr/sc` pair or AMO instructions (zalrsc- or zaamo-extension)