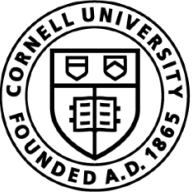


Cornell Bowers CIS
Computer Science

CS3410: Computer Systems and Organization

LEC24: Atomicity and Synchronization

Dr. Kevin Laeufer
Wednesday, November 19, 2025

Credits: Alvisi, Bala, Bracy, Garcia, Guidi, Kao, Laeufer, Martin, McKee, Sampson, Sirer, Van Renesse, Weatherspoon

Poll: Special Topics Lectures

PollEv.com/cs3410

Plan for the next 2 lectures.

- Review: Threads and Race Conditions
- Synchronization:
  - LR/SC
  - Atomic Increment,
  - Critical Section
  - Mutex
  - Memory Consistency
- Practical Programming with Threads

Review: Threads

Threads are **separate tasks** within the same process.

- Shared: code, data, heap, page table, files
- Private: registers, stack, PC

Processes, on the other hand:

- Use separate page tables.
- Need to be isolated (trust boundary).

Review: Race conditions

Occur when two threads are accessing the same memory location and at least one access is a write.
(two concurrent reads are fine!)

Challenges of Race Conditions

- Races are intermittent, may occur rarely or in certain scenarios
- They often appear to happen ***randomly***

Program is correct *only* if *all possible* schedules are safe

Example Race Condition: 2 threads trying to increment `x`

Review: OS vs. User-Level Threads

OS Threads

- preemptive context switch allows OS to interrupt thread at any time
- PC, SP, stack, registers managed by OS
- can use blocking system call
- each thread can run on a different CPU core

User-Level Threads

- thread must yield before context switch
- PC, SP, stack, registers managed by library
- blocking system call would block all other threads
- all threads share a single CPU core

Poll: Race conditions with user-level threads.

PollEv.com/cs3410

PollEV Question

Can hardware help solve race conditions?

A. Yes, without the programmer's involvement
B. Yes, by offering helpful tools to programmers
C. Yes, and race conditions cannot be solved without hardware support
D. No, the programmer is on their own
E. I don't know.

PollEv.com/cs3410

Hardware Support for Synchronization

Atomic read & write memory operation

- Between read & write: *no writes to that address*

Many atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and swap (x86, ARM deprecated)
- load reserved / store conditional
  (RISC-V, MIPS, ARM, PowerPC, DEC Alpha, …)

Synchronization in RISC-V

Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`

"I want the value at address X. Also, start monitoring any writes to this address."

Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

"If no one has changed the value at address X since the LR.{W,D}, perform this store and tell me it worked."

- Data at location **has NOT been written to** since the LR?
  - **SUCCESS:**
    - Performs the store (value in rs2 written to address in rs1)
    - Returns 0 in rd
- Data at location **has been written to** since the LR?
  - **FAILURE:**
    - Does not perform the store
    - Returns 1 in rd

Using LR/SC to create Atomic Increment

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

i++

↓

```
LW t0, (s1)
ADDI t0,t0,1
SW t0, (s1)
```

atomic(i++)

↓

```
try: LR.W.aqrl t0, (s1)
     ADDI t0,t0,1
     SC.W.aqrl t0, t0,(s1)
     BNEZ t0, try
```

Value in memory written between LR and SC ?

□ SC returns 1 in `t0` □ go back & try again

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1					
2					
3					
4					
5					
6					
7					
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2					
3					
4					
5					
6					
7					
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3					
4					
5					
6					
7					
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0, t0, 1		1	0	0
4					
5					
6					
7					
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0, t0, 1		1	0	0
4		ADDI t0, t0, 1	1	1	0
5					
6					
7					
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0, t0, 1		1	0	0
4		ADDI t0, t0, 1	1	1	0
5	SC.W t0, t0,(s1)		**0**	1	1
6					
7					
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0, t0, 1		1	0	0
4		ADDI t0, t0, 1	1	1	0
5	SC.W t0, t0,(s1)		**0**	1	1
6	BNEZ t0, try		0	1	1
7	*Success!*				
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0, t0, 1		1	0	0
4		ADDI t0, t0, 1	1	1	0
5	SC.W t0, t0,(s1)		**0**	1	1
6	BNEZ t0, try		0	1	1
7	*Success!*	SC.W t0, t0, (s1)	0	**1**	1
8					

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0, t0, 1		1	0	0
4		ADDI t0, t0, 1	1	1	0
5	SC.W t0, t0,(s1)		**0**	1	1
6	BNEZ t0, try		0	1	1
7		SC.W t0, t0, (s1)	0	**1**	1
8		BNEZ t0, try	0	1	1

Success!

Failure!

Atomic Increment in Action

- Load Reserved: `LR.{W,D}.aqrl rd, (rs1)`
- Store Conditional: `SC.{W,D}.aqrl rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1	try: LR.W t0, (s1)		0		0
2		try: LR.W t0, (s1)		0	0
3	ADDI t0,				0
4					0
5	SC.W t0,				1
6	BNEZ t0,				1
7		SC.W t0, t0, (s1)	0	1	1
8		BNEZ t0, try	0	1	1

Thread B will now retry until its increment is visible!

Failure!

PollEV Question

Another HW synchronization primitive is CAS (compare-and-swap).
Which of the following is true?

A. LR/SC is more *complex* than CAS

B. LR/SC can be used to implement CAS

C. RISC-V does not implement CAS b/c
CAS is better suited to a CISC architecture

D. A & B

E. B & C

PollEv.com/cs3410

```
CAS new, old, (addr)
```

```
if (*addr == old) {
    *addr = new;
    new = 0; //success
} else {
    new = 1; //fail
}
```

Critical Sections

- Create atomic version of every instruction?
  - **NO**: Does not scale *or solve the problem*

- To eliminate races: identify *Critical Sections*
  - Places in code where shared state is read and written
  - ***Only 1 thread gets to execute at a time***
  - Others *wait their turn*

Critical Section!

```
...
tmp = *cur_score;                                  //read shared var
*cur_score = update_score(tmp);                    //write shared var
if (tmp > *high_score) { *high_score = tmp; } //r/w shared var
...
```

Critical Sections

- Create atomic version of every instruction?
  - **NO**: Does not scale *or solve the problem*

- To eliminate races: identify *Critical Sections*
  - Places in code where shared state is read and written
  - ***Only 1 thread gets to execute at a time***
  - Others *wait their turn*

```
cs_enter();
tmp = *cur_score;                                  //read shared var
*cur_score = update_score(tmp);                    //write shared var
if (tmp > *high_score) { *high_score = tmp; } //r/w shared var
cs_exit();
```

Mutual Exclusion Lock (Mutex)

- Implementation of *cs_enter()* and *cs_exit()*

```
lock = 0; // global variable
          // 0 means lock is free
          // 1 means lock is taken
```

```
cs_enter(lock);
tmp = *cur_score;
*cur_score = update_score(tmp);
if (tmp > *high_score) {
 *high_score = tmp;
}
cs_exit(lock);
...
```

Thread 0

Atomically:
Wait until **lock** is 0, then set to 1

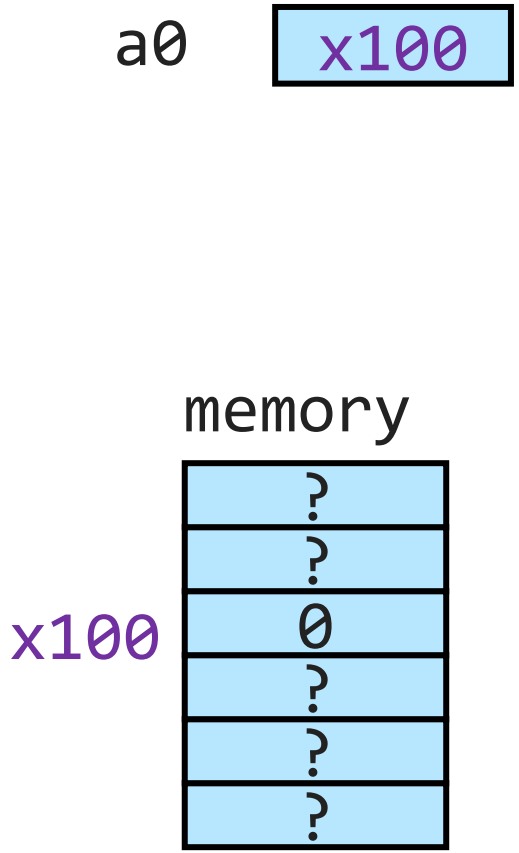
I am the **ONLY THREAD** running this code!

Set **lock** to 0

Mutex from LR and SC

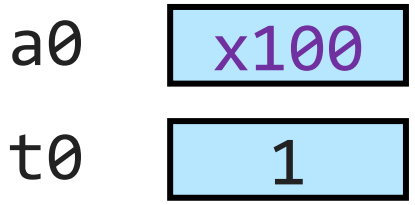
Start Here

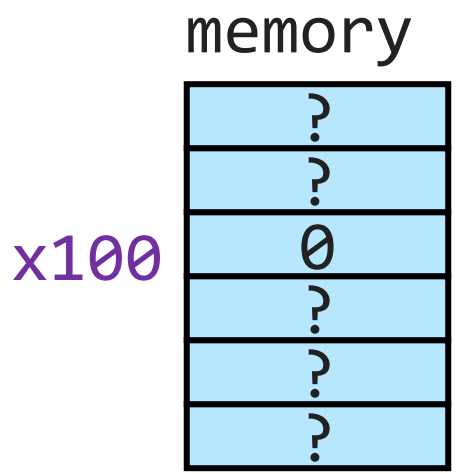
```
lock = 0;  // global variable @ address 0x100
         // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {




}

mutex_unlock(int *lockAddr) {
    ...
}
```



Mutex from LR and SC

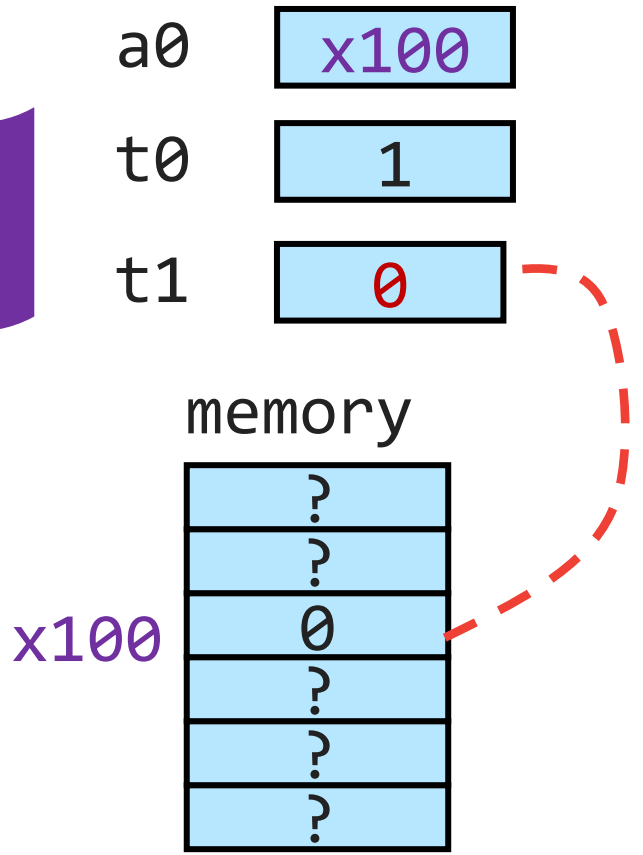
```
lock = 0;  // global variable @ address 0x100
         // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1




}

mutex_unlock(int *lockAddr) {
    ...
}
```





Mutex from LR and SC

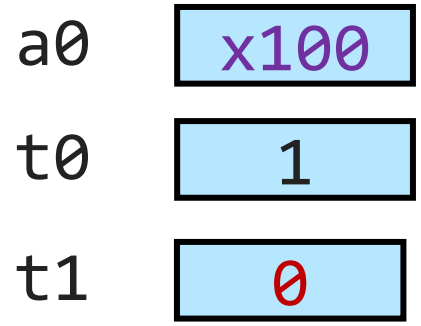
```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)



}

mutex_unlock(int *lockAddr) {
    ...
}
```

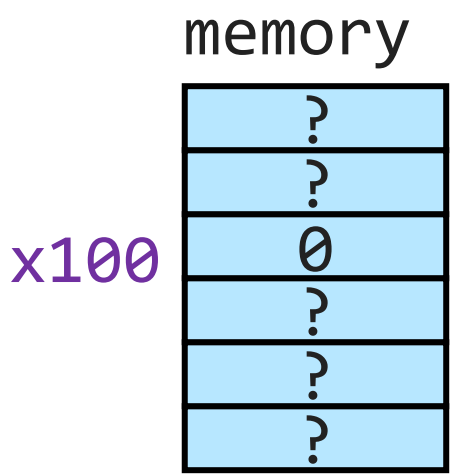


Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy



}

mutex_unlock(int *lockAddr) {
    ...
}
```

Reserved: @x100

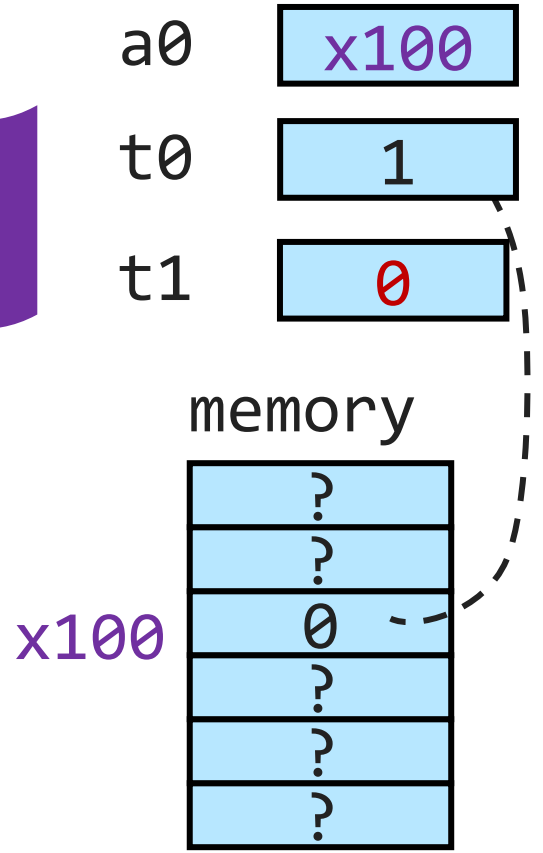




Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W.aqrl t0, t0, (a0)

}

mutex_unlock(int *lockAddr) {
    ...
}
```

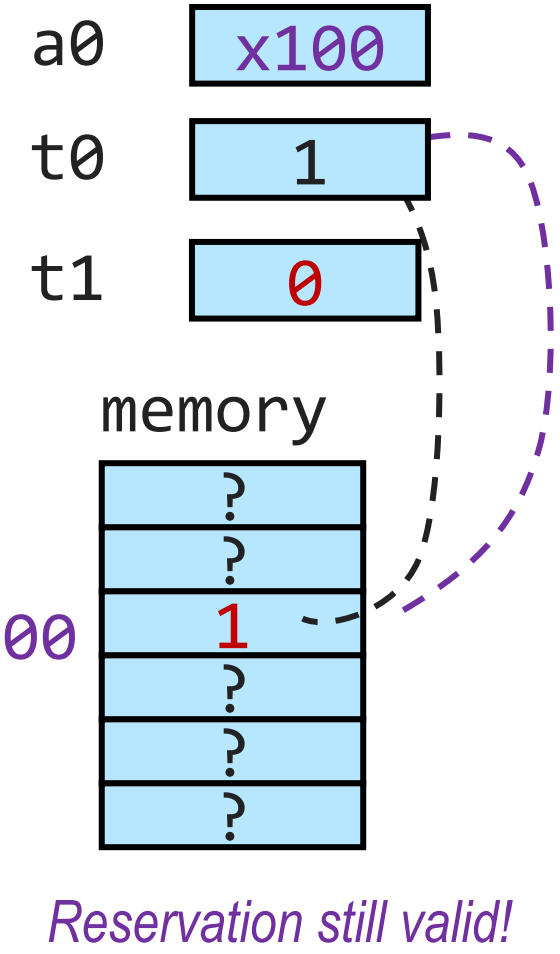
Reserved: @x100



Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W.aqrl t0, t0, (a0)

}

mutex_unlock(int *lockAddr) {
    ...
}
```

Reserved: @x100

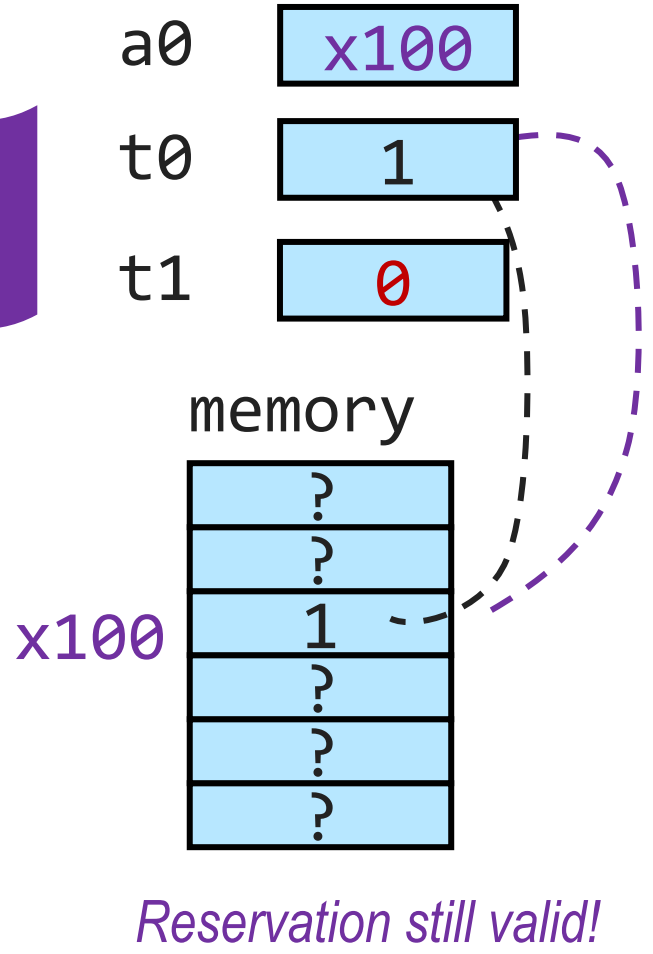


Reservation still valid!

Mutex from LR and SC

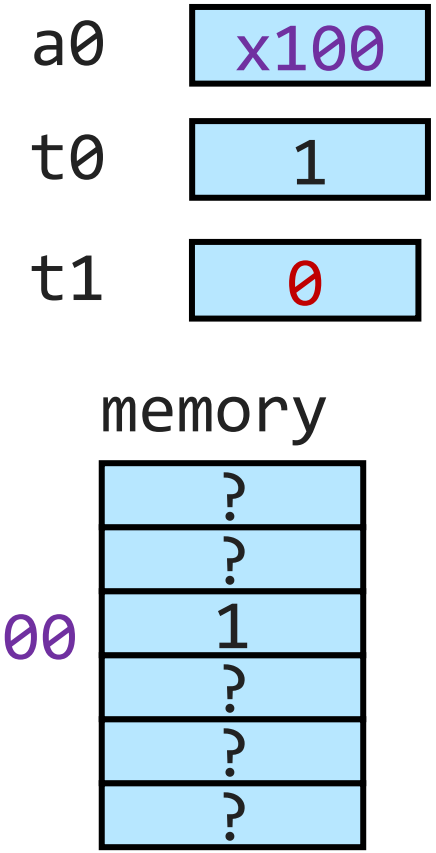
```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W.aqrl t0, t0, (a0)

}

mutex_unlock(int *lockAddr) {
    ...
}
```

Reserved: @x100



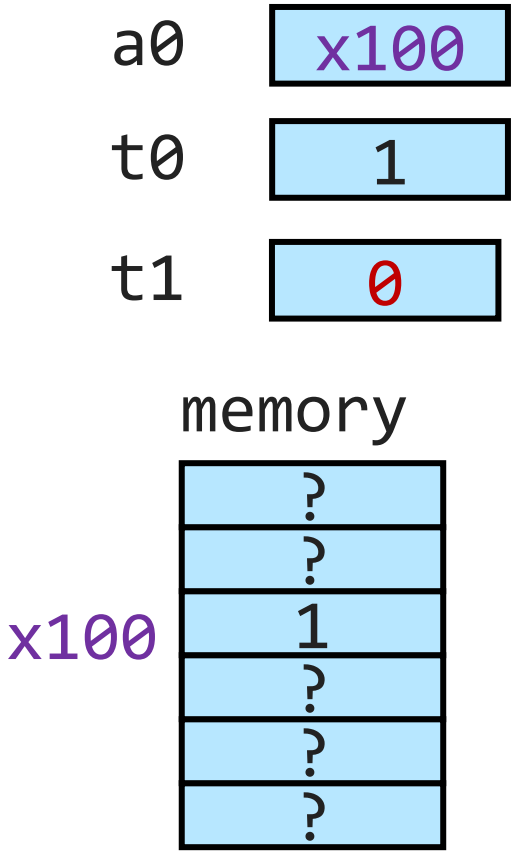
Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W.aqrl t0, t0, (a0)
          BNEZ t0, t_and_s // t0 != 0 => lost the race
}

mutex_unlock(int *lockAddr) {
    ...
}
```



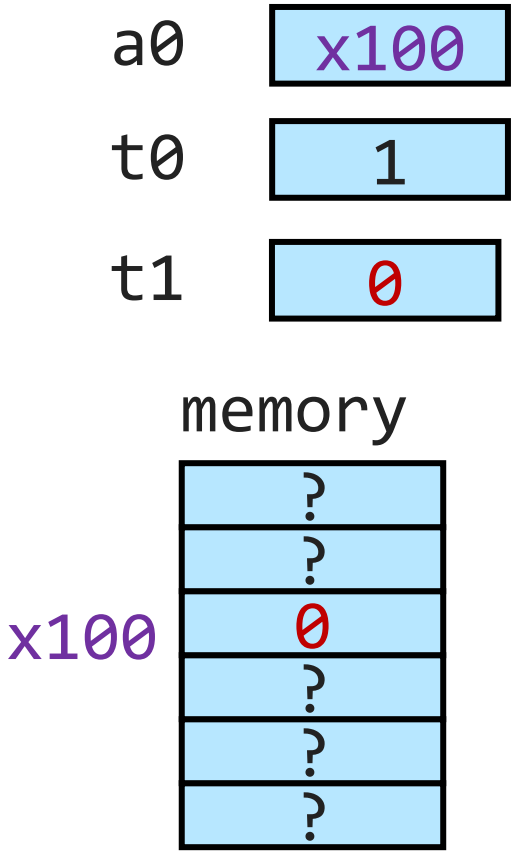
Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W.aqrl t0, t0, (a0)
          BNEZ t0, t_and_s // t0 != 0 => lost the race
}

mutex_unlock(int *lockAddr) {
    SW x0, (a0);
}
```



Mutex from LR and SC

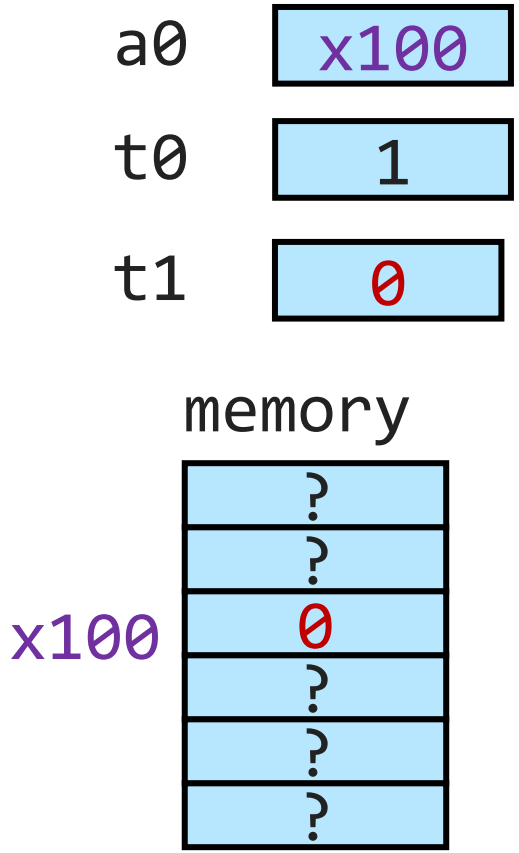
```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) { // a spin lock
 t_and_s: LI t0, 1
          LR.W.aqrl t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W.aqrl t0, t0, (a0)
          BNEZ t0, t_and_s // t0 != 0 => lost the race
}

mutex_unlock(int *lockAddr) {
    SW x0, (a0);
}
```



Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
        // 0 => free; 1 => taken
mutex_lock(int *lockAddr) { // a spin lock
  while(t_and_s(lockAddr)){}
}
int t_and_s(int *lockAddr) {
  old = *lockAddr;
  *lockAddr = 1;
  return old;
}
mutex_unlock(int *lockAddr) {
   *lockAddr = 0;
}
```

LR.W Atomic
SC.W



x86 provides a BTS “test and set” instruction

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2							
3							
4							
5							
6							
7							
8							

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2	LR.W t1,(a0)		1	**0**	1		0
3		LR.W t1,(a0)	1	0	1	**0**	0
4	BNEZ t1, try	BNEZ t1, try	1	0	1	0	0

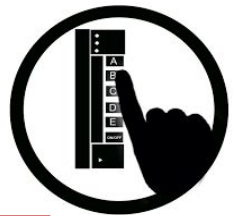
What will happen?

(A) Thread A moves on to SC

(B) Thread B moves on to SC

(C) A & B both move on to SC

(D) both threads go back to try



PollEv.com/cs3410

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2	LR.W t1,(a0)		1	**0**	1		0
3		LR.W t1,(a0)	1	0	1	**0**	0
4	BNEZ t1, try	BNEZ t1, try	1	0	1	0	0
5							
6							
7							
8							

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2	LR.W t1,(a0)		1	**0**	1		0
3		LR.W t1,(a0)	1	0	1	**0**	0
4	BNEZ t1, try	BNEZ t1, try	1	0	1	0	0
5		SC.W t0,t0,(a0)			**0**	0	**1**
6	SC.W t0, t0,(a0)		**1**	0	0	0	1
7							
8							

Failure!

Success!

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2	LR.W t1,(a0)		1	**0**	1		0
3		LR.W t1,(a0)	1	0	1	**0**	0
4	BNEZ t1, try	BNEZ t1, try	1	0	1	0	0
5		SC.W t0, t0,(a0)			**0**	0	**1**
6	SC.W t0, t0,(a0)		**1**	0	0	0	1
7	BNEZ t0, try	BNEZ t0, try	1	0	0	0	1
8							

Both threads check to see if they successfully claimed the lock

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Thread B enters Critical section

Thread A tries again…

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2	LR.W t1,(a0)		1	**0**	1		0
3		LR.W t1,(a0)	1	0	1	**0**	0
4	BNEZ t1, try	BNEZ t1, try	1	0	1	0	0
5		SC.W t0, t0,(a0)			**0**	0	**1**
6	SC.W t0, t0,(a0)		**1**	0	0	0	1
7	BNEZ t0, try	BNEZ t0, try	1	0	0	0	1
8	try: LI t0, 1	Critical section					

2 threads attempt to grab the lock

```
mutex_lock(int *lockAddr)
```

```
try: LI t0, 1
LR.W.aqrl t1, (a0)
BNEZ t1, try
SC.W.aqrl t0, t0, (a0)
BNEZ t0, try
```

Thread B enters Critical section

Thread A tries again…

lock is taken! Thread A must continue trying until lock is free

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2	LR.W t1,(a0)		1	**0**	1		0
3		LR.W t1,(a0)	1	0	1	**0**	0
4	BNEZ t1, try	BNEZ t1, try	1	0	1	0	0
5		SC.W t0, t0,(a0)			**0**	0	**1**
6	SC.W t0, t0,(a0)		**1**	0	0	0	1
7	BNEZ t0, try	BNEZ t0, try	1	0	0	0	1
8	try: LI t0, 1	Critical section	1	0	0	0	1
9	LR.W t1,(a0)	Critical section	1	1	0	0	1

Problem Solved?

```
lock = 0;
x = 0;
```

Thread 1

```
for (i = 0; i < 5; i++) {
   mutex_lock(&lock);
   x = x + 1;
   mutex_unlock(&lock);
}
```

a) 6
b) 8
c) 10
d) Could be any of the above
e) Couldn't be any of the above

Thread 2

```
for (i = 0; i < 5; i++) {
   mutex_lock(&lock);
   x = x + 1;
mutex_unlock(&lock);
}
```

`lock` and `x` are global variables. What will the value of `x` be after both loops finish?

Memory Consistency

Thread 1

```
for (i = 0; i < 5; i++) {
   mutex_lock(&lock);
   x = x + 1;
   mutex_unlock(&lock);
}
```

```
s: LI t0, 1
   LR.W.aqrl t1, (&lock)
   BNEZ t1, s
   SC.W.aqrl t0, t0, (&lock)
   BNEZ t0, s
```

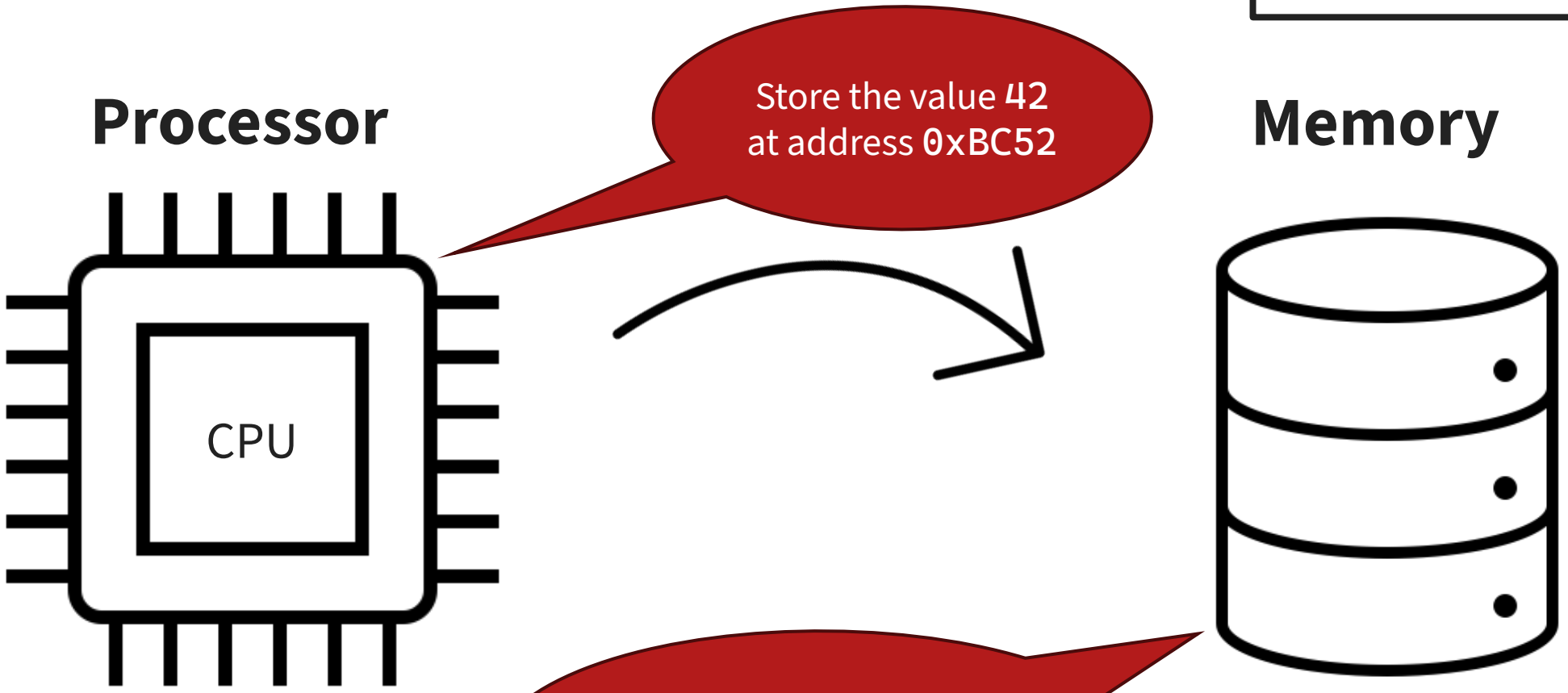
```
SW x0, (&lock);
```

What prevents the CPU from loading the value of &x before the lock is acquired?

What prevents the CPU from storing the new value of &x after the lock is released?

A Mental Model of Memory

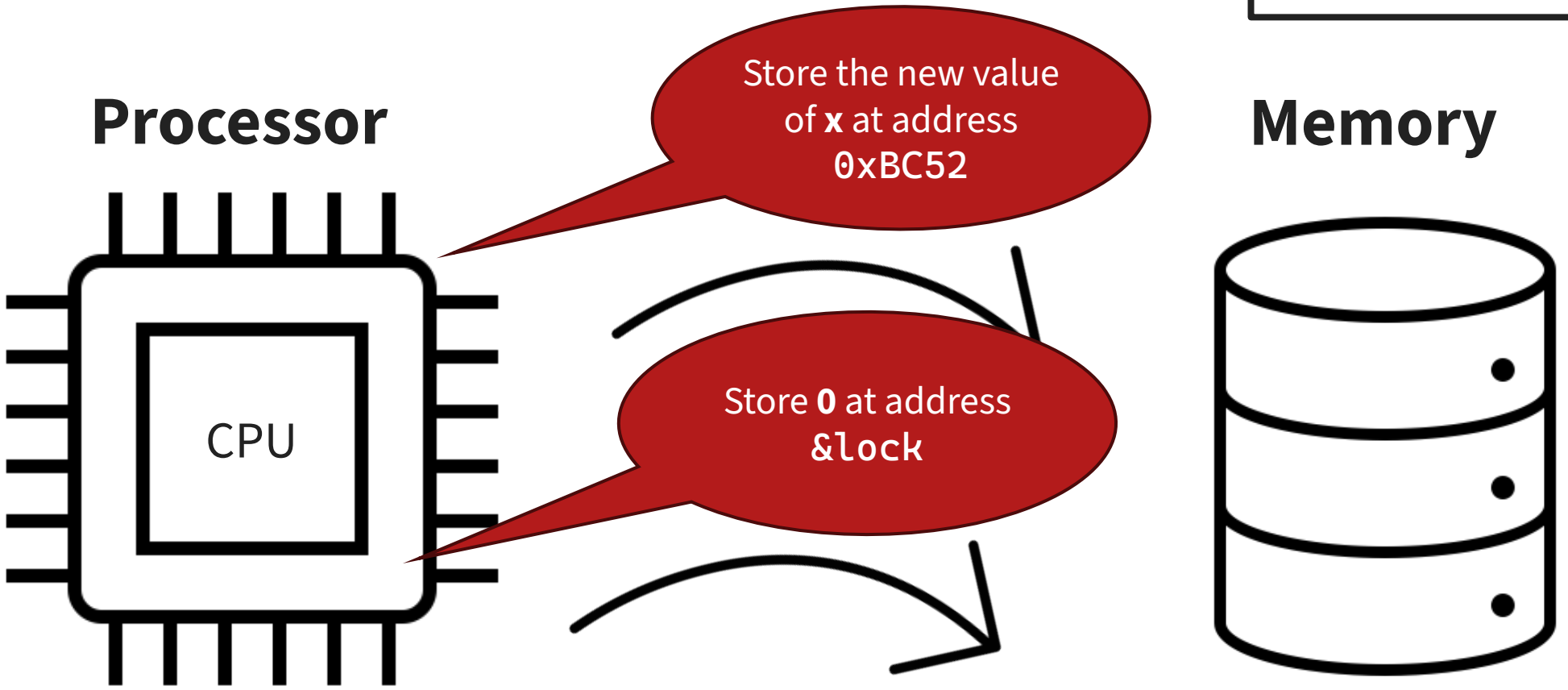
from lecture 5



$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

A Mental Model of Memory

from lecture 5




```
uint8_t mem[SIZE];
```

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

Memory Consistency

- A memory consistency model describes how loads and stores can be reordered within a thread and across CPU cores.
- Reordering cannot be detected in a single thread and is important for performance. However, it can lead to correctness issues with multiple threads.
- All our examples assume that instructions in a single thread are executed strictly in-order. This model is called sequential consistency.
- We use `.aqrl` as a suffix to `lr` and `sc` to avoid reordering.
- Technically, our lock release code does not enforce ordering and is thus incorrect.

Condition Variables

- We want to wait until data is available:

```
// lock protects
// data_valid and data
while(true) {
    mutex_lock(&lock);
    if(data_valid) {
        // store data
        data_valid = 0;
    }
    mutex_unlock(&lock);
}
```

```
mutex_lock(&lock);
while(data_valid == 0) {
    cond_wait(&cond, &lock);
}
// store data
data_valid = 0;
mutex_unlock(&lock);
```

Condition Variables

- We could just release and reacquire the lock in a loop and check our variable. This leads to a lot of costly memory operations.
- Instead, we want to tell our operating system to let our thread sleep until a change has happened.
- Condition variables allow a receiver to wait for a signal and a sender to wake up the receiver when it is time.
- Spurious wakeup: it is not guaranteed that the value actually has changed. E.g., because there were two changes: $0 \rightarrow 1 \rightarrow 0$
- You will implement a version of condition variables in assignment 11.

Synchronization Variations

Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

Semaphores

- N threads can hold lock at the same time
- Used for “resource counting”

Monitors

- Concurrency-safe data structure with 1 mutex
- All operations on monitor acquire/release mutex
- One thread in the monitor at a time

Curious about these? *Take CS 4410!*

CS 3410 Takeaway: HW provides the primitives (e.g., LR/SC) to support thread-level synchronization operations.

Summary

- Threads are great for improved performance.
- Avoiding data races is difficult.
- Critical section: program code that needs to happen atomically
- Lock allows only one thread to enter the critical section.
- Hardware provides synchronization primitives such as **LR** and **SC** instructions to efficiently and **correctly** (!) implement locks.