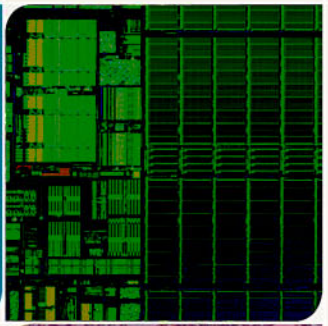
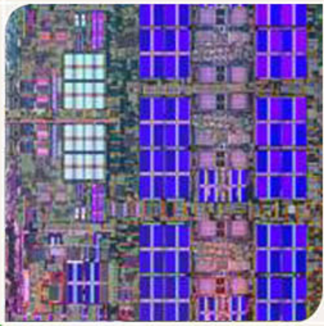
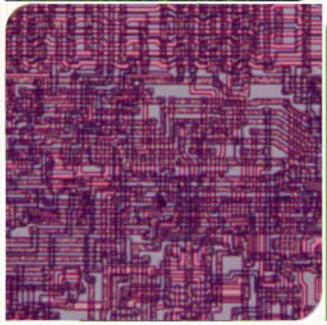
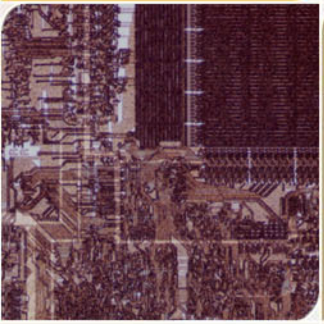





CS4290/CS6290

Fall 2011

Prof. Hyesoon Kim

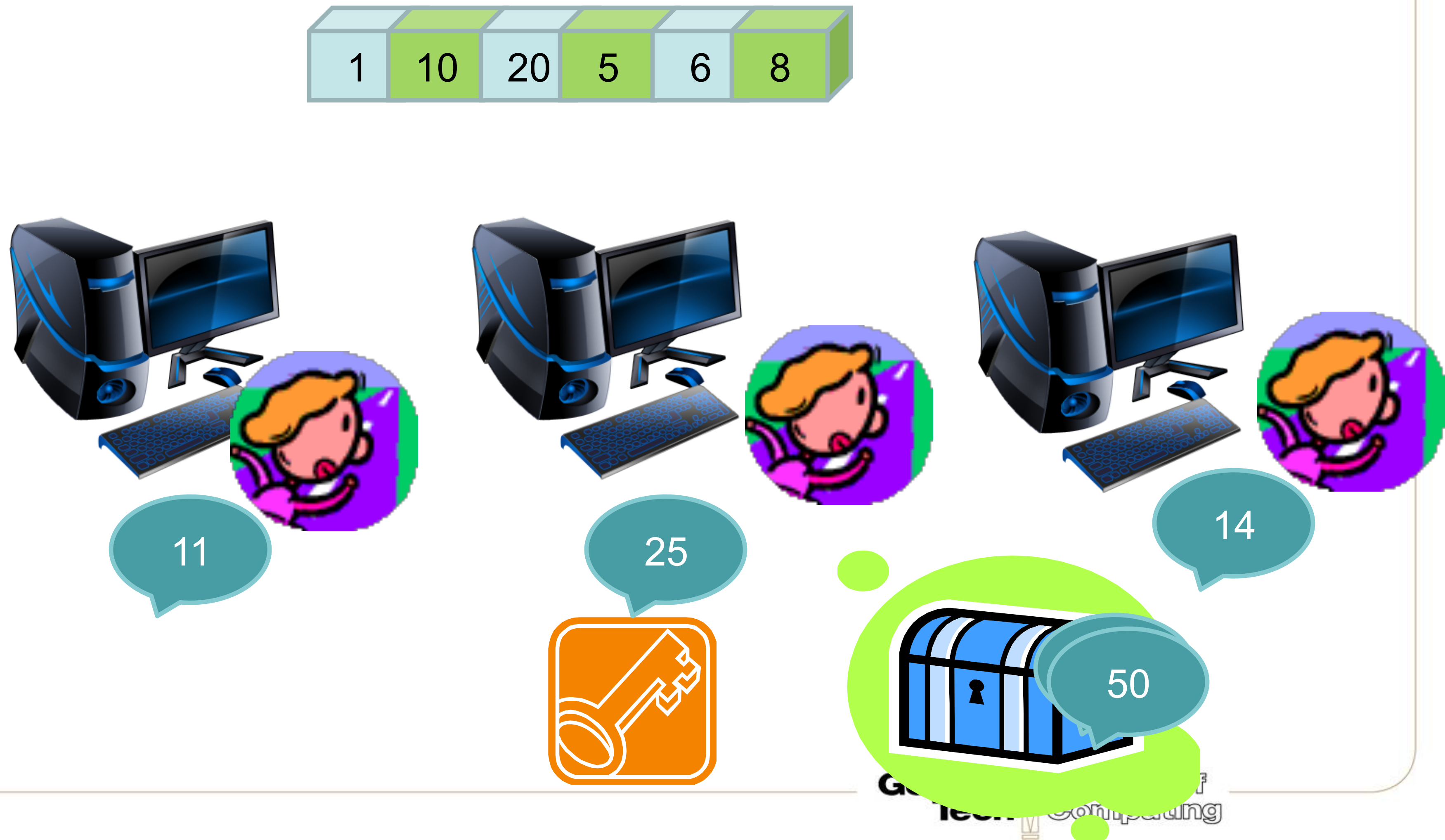


Georgia Tech College of Computing

Thanks to Prof. Loh & Prof. Prvulovic

PARALLEL PROGRAMMING & HARDWARE IMPLEMENTATION

Lock Example (Review)



Synchronization

- Shared counter/sum update example
  - Use a mutex variable for mutual exclusion
  - Only one processor can own the mutex
    - Many processors may call lock(), but only one will succeed (others block)
    - The winner updates the shared sum, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)

Georgia Tech College of Computing

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - Need a way to ***atomically*** see that the mutex is 0 ***and*** set it to 1

Atomic Read-Update Instructions

- Atomic exchange instruction
  - E.g., EXCH R1,78(R2) will swap content of register R1 and mem location at address 78+R2
  - To acquire a mutex, 1 in R1 and EXCH
    - Then look at R1 and see whether mutex acquired
    - If R1 is 1, mutex was owned by somebody else and we will need to try again later
    - If R1 is 0, mutex was free and we set it to 1, which means we have acquired the mutex
- Other atomic read-and-update instructions
  - E.g., Test-and-Set

LL & SC Instructions

- Atomic instructions OK, but specialized
  - E.g., SWAP can not atomically inc a counter
- Idea: provide a pair of linked instructions
- A load-linked (LL) instruction
  - Like a normal load, but also remembers the value in a special “link” register
- A store-conditional (SC) instruction
  - Like a normal store, but fails if its value is not the same as that in the link register
  - Returns 1 if successful, 0 on failure
- Writes by other processors snooped
  - If value matches link register, clear link register

Using LL & SC

Swap R4 w/ 0(R1)

Atomic Exchange

```
swap:  mov     R3, R4
       ll      R2,0(R1)
       sc      R3,0(R1)
       beqz    R3,swap
       mov     R4,R2
```

Simple Swap's problem

- MSI protocol :

Acquire a lock

P1	P2	P3

- – P1 SWAP "1" mem[A]
- – P2 SWAP "1" mem[A]
- – P3 SWAP "1" mem[A]
- – P2 SWAP "1" mem[A]
- – P3 SWAP "1" mem[A]
- – P2 SWAP "1" mem[A]

So many invalidations!!

Implementing Locks

- A simple swap (or test-and-set) works
  - But causes a lot of invalidations
    - Every write sends an invalidation
    - Most writes redundant (swap 1 with 1)
- More efficient: test-and-swap
  - Read, do swap only if 0
    - Read of 0 does not guarantee success (not atomic)
    - But if 1 we have little chance of success
  - Write only when good chance we will succeed

Example: Test and Test and Set

```
try:    mov     R3,#1
        ll      R2,0(R1)
        sc      R3,0(R1)
        bnez    R2,try
        beqz    R3,try
```

$\Rightarrow$

```
try:    mov     R3,#1
        ll      R2,0(R1)
        bnez    R2,try
        sc      R3,0(R1)
        beqz    R3,try
```

Using LL & SC

Swap R4 w/ 0(R1)

Test if 0(R1) is zero, set to one

Atomic Exchange

```
swap:   mov     R3, R4
        ll      R2,0(R1)
        sc      R3,0(R1)
        beqz    R3,swap
        mov     R4,R2
```

Atomic Test&Set

```
t&s:    mov     R3,1
        ll      R2,0(R1)
        sc      R3,0(R1)
        bnez    R2,t&s
        beqz    R3,t&s
```

Atomic Add to Shared Variable

```
upd:    ll      R2,0(R1)
        add     R3,R2,R4
        sc      R3,0(R1)
        beqz    R3,upd
```

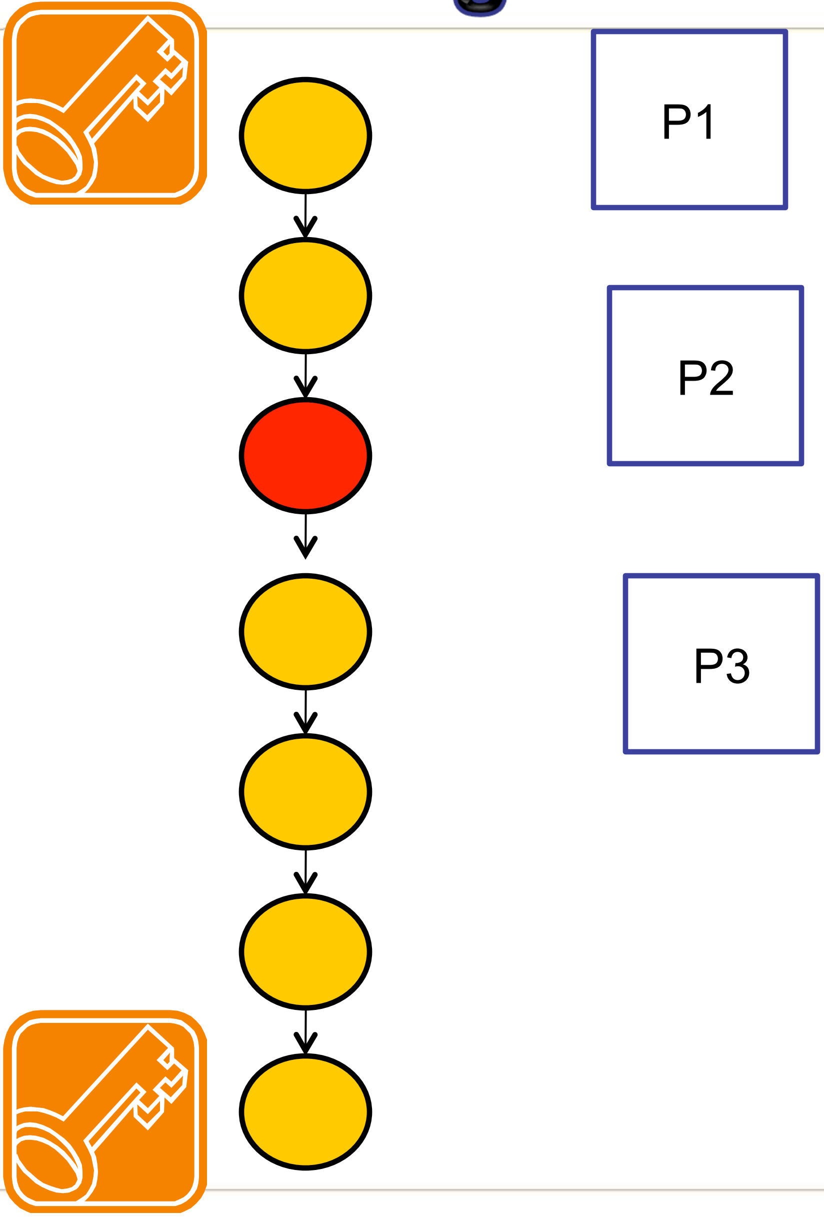
Large-Scale Systems: Locks

- Contention even with test-and-test-and-set
  - Every write goes to many, many spinning procs
  - Making everybody test less often reduces contention for high-contention locks but hurts for low-contention locks
  - Solution: exponential back-off
    - If we have waited for a long time, lock is probably high-contention
    - Every time we check and fail, double the time between checks
      - Fast low-contention locks (checks frequent at first)
      - Scalable high-contention locks (checks infrequent in long waits)
  - Special hardware support

Barrier Synchronization

- All must arrive before any can leave
  - Used between different parallel sections
- Uses two shared variables
  - A counter that counts how many have arrived
    - atomically dec a counter
  - A flag that is set when the last processor arrives

Over using locks



```
Acquire lock
While (node) {
        If (node->data > 10)
        node->data = new_data;
        node = node->next
}
Release lock
```

Transactional Memory

A group of load and store instructions to execute in an atomic way.

Trying to avoid locks

- Hardware transactional memory
- Begin/End
- Commit or Abort
- Commits: All of the loads and stores appear to have run atomically with respect to other transactions
- How? Keep a record (in a hardware or in a memory)

Review Question

- MSI protocol : cache block size is 4B
  - P1 LDB mem[A]
  - P2 STB mem[A]
  - P3 LDB mem[A+1]
  - P1 STB mem[A+2]
  - P2 STB mem[A]
  - P3 LDB mem[A+1]

	P1	P2	P3

MEMORY CONSISTENCY

Memory Consistency

- Coherence is about order of accesses to the same address
- Consistency is about order of accesses to different addrs

Program order

```
Proc 1    Proc 2
ST 1->D   LD F
ST 1->F   LD D
```

$\Rightarrow$

Execution order

```
Proc 1    Proc 2
ST D      LD D
ST F      LD F
```

Proc 2 can reorder loads

- Possible outcomes on Proc 2 in program order
  - (F,D) can be (0,0), (0,1), (1,1)
- Execution order can also give (1,0)
  - Exposes instruction reordering to programmer
  - We need something that makes sense intuitively

Memory Consistency

- But first: why would we want to write code like that?
  - If we never actually need consistency, we don't care if it isn't there
- Example: flag synchronization
  - Producer produces data and sets flag
  - Consumer waits for flag to be 1, then reads data

Source Code

```
Proc 1      Proc 2
D=Val1;     while(F==0);
F=1;        Val2=D;
```

$\Rightarrow$

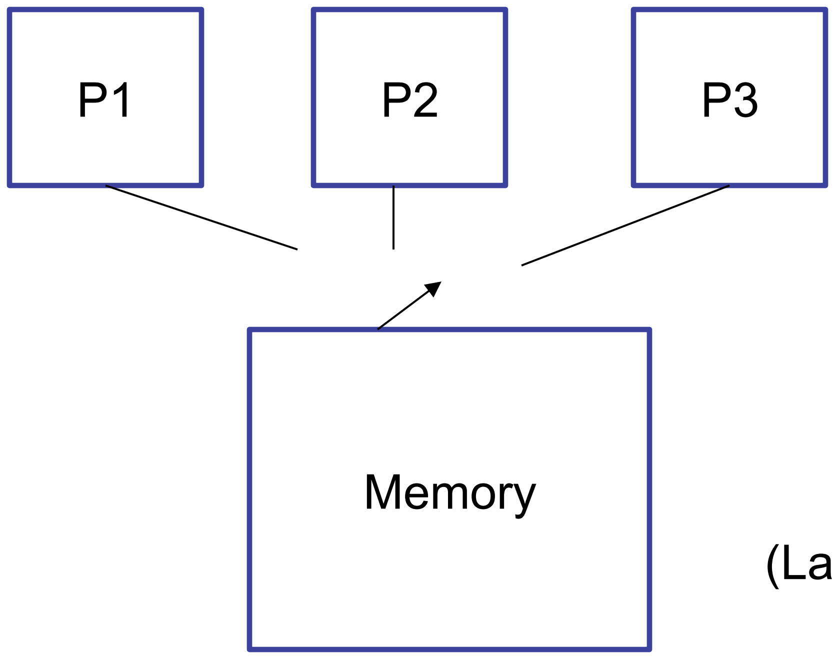
Assembler Code

```
Proc 1     Proc 2
ST D       wait:  LD F
ST F              BEQZ F,wait
                  LD D
```

- Now we can have the following situation
  - Proc 2 has cache miss on F, predicts the branch not taken, reads D
  - Proc 1 writes D, writes F
  - Proc 2 gets F, checks, sees 1, verifies branch is correctly predicted

Sequential Consistency (SC)

- The result of any execution should be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved
  - The same interleaved order seen by everybody

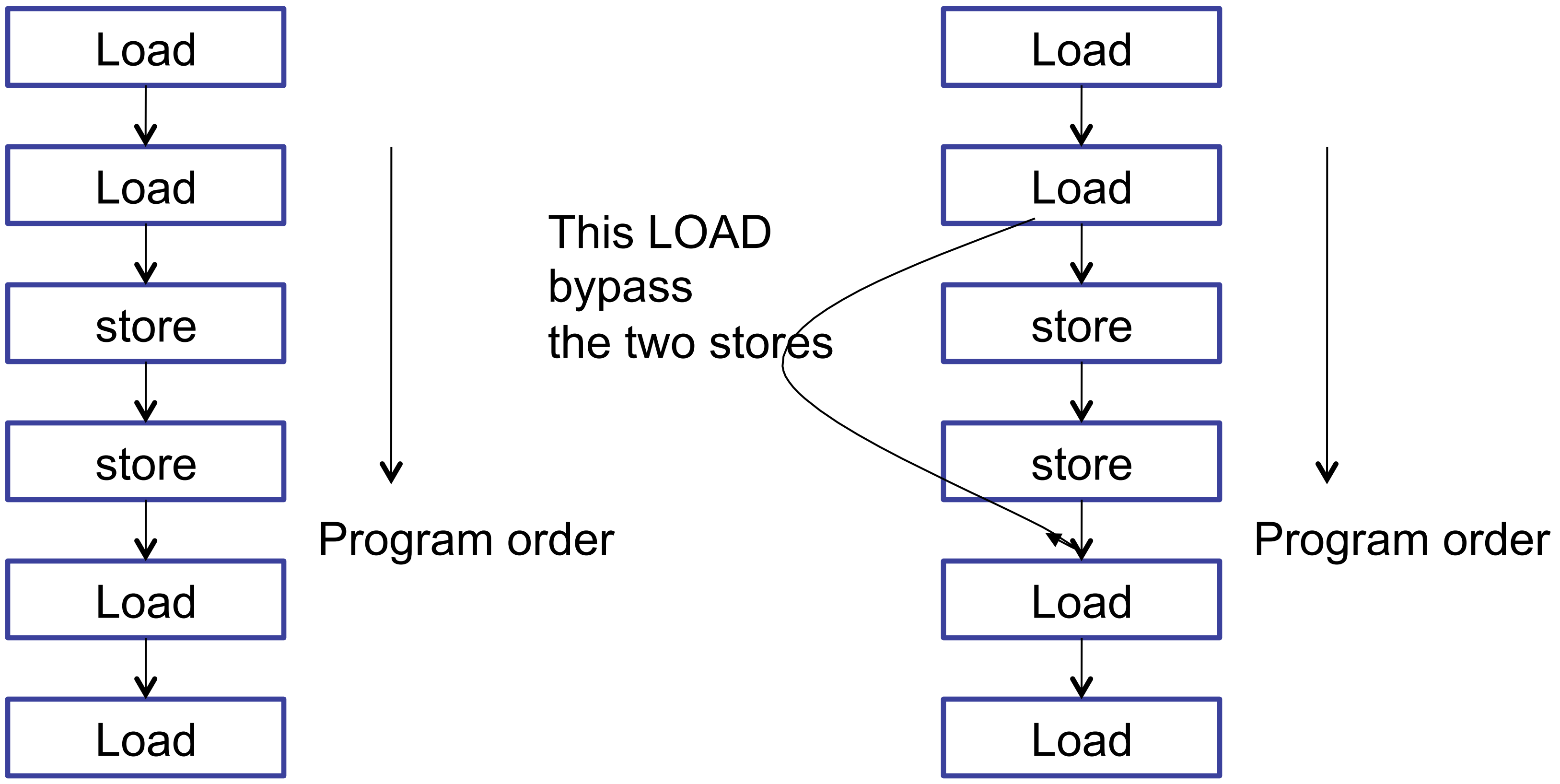


(Lamport)

Hardware support for SC

- Simple implementation
  - A processor issues next access only when its previous access is complete
  - Write completion acknowledgement, invalidation acknowledgment
  - Sloooow!
- A better implementation
  - Processor issues accesses as it sees fit, but detects and fixes potential violations of sequential consistency

Relaxed Consistency Models



Sequential consistency

Processor consistency (PC)
Loads are allowed to by pass stores

Relaxed Consistency Models

- Two kinds of memory accesses
  - Data accesses and synchronization accesses
- Synchronization accesses determine order
  - Data accesses ordered by synchronization
  - Any two of accesses to the same variable in two different processes, such that at least one of the accesses is a write, are always ordered by synchronization operations
- Good performance even in simple implementation
  - Sequential consistency for sync accesses
  - Data accesses can be freely reordered (except around sync accesses)

Relaxed Consistency Models

- Data Races
  - When data accesses to same var in different procs are not ordered by sync
  - Most programs are data-race-free (so relaxed consistency models work well)
- There are many relaxed models
  - Weak Consistency, Processor Consistency, Release Consistency, Lazy Release Consistency
  - All work just fine for data-race-free programs
  - But when there are data races, more relaxed models $\Rightarrow$ weirder program behavior

Data Race?

- Two concurrent accesses to a shared location, at least one of them for writing.
  - BUG!!

$$\begin{tikzcd}
\underline{\text{Thread 1}} & \underline{\text{Thread 2}} \\
X{+}{+} \arrow[dr, leftrightarrow] & T=Y \\
Z=2 & T=X
\end{tikzcd}$$