WELCOME
TO
15-213
18-213
15-513

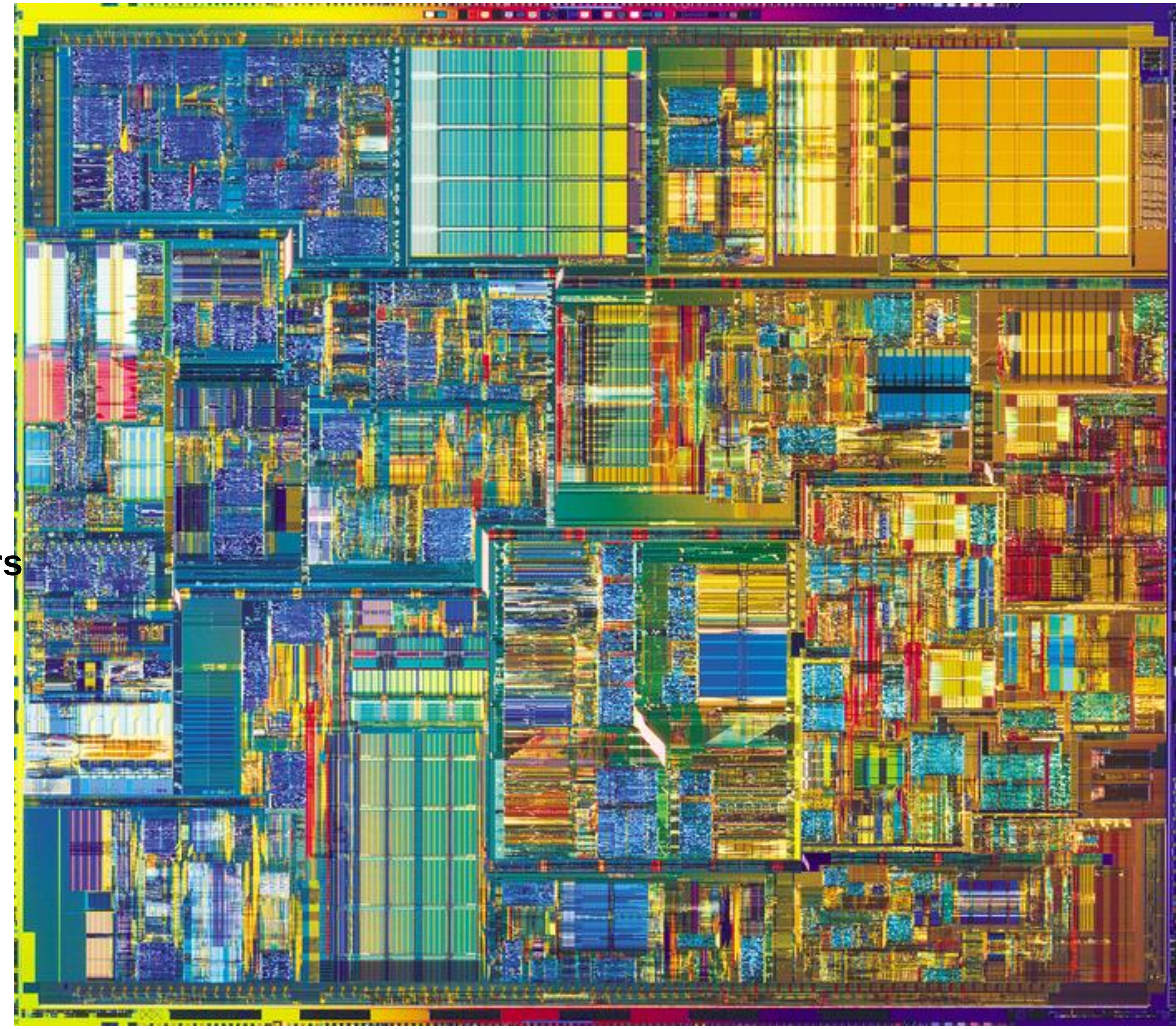
Thread-Level Parallelism

15-213 / 18-213 / 14-513 / 15-513: Introduction to Computer Systems
26th Lecture, April 25, 2019

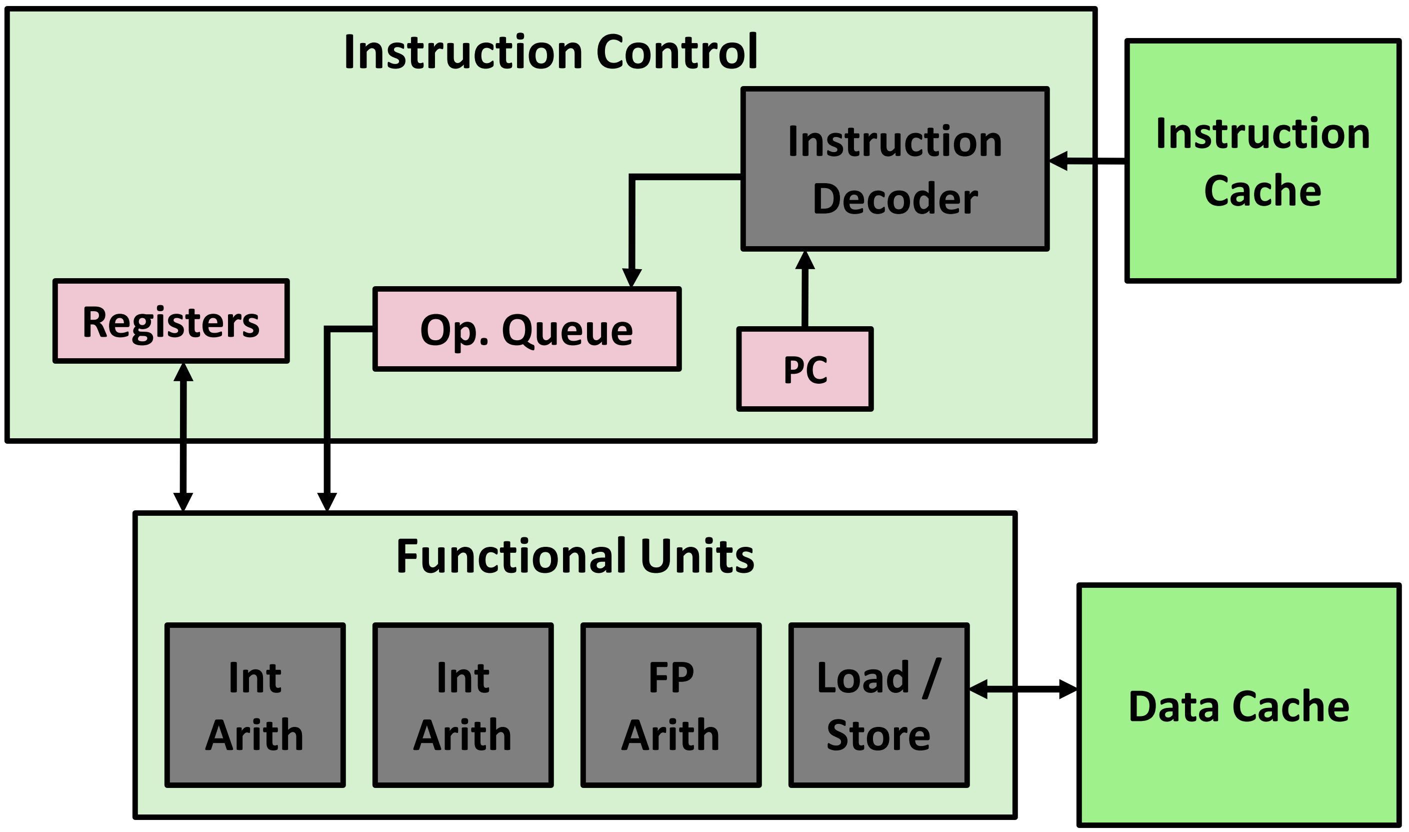
Today

- **Parallel Computing Hardware**
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core
- **Thread-Level Parallelism**
  - Splitting program into independent tasks
    - Example: Parallel summation
    - Amdahl's Law
- **Cache Coherence & Memory Consistency**
  - What happens when multiple threads are reading & writing shared state

- **Willamette core**
- **180 nm process**
- **217 mm² die size**
- **42,000,000 transistors**
- 2000-2008

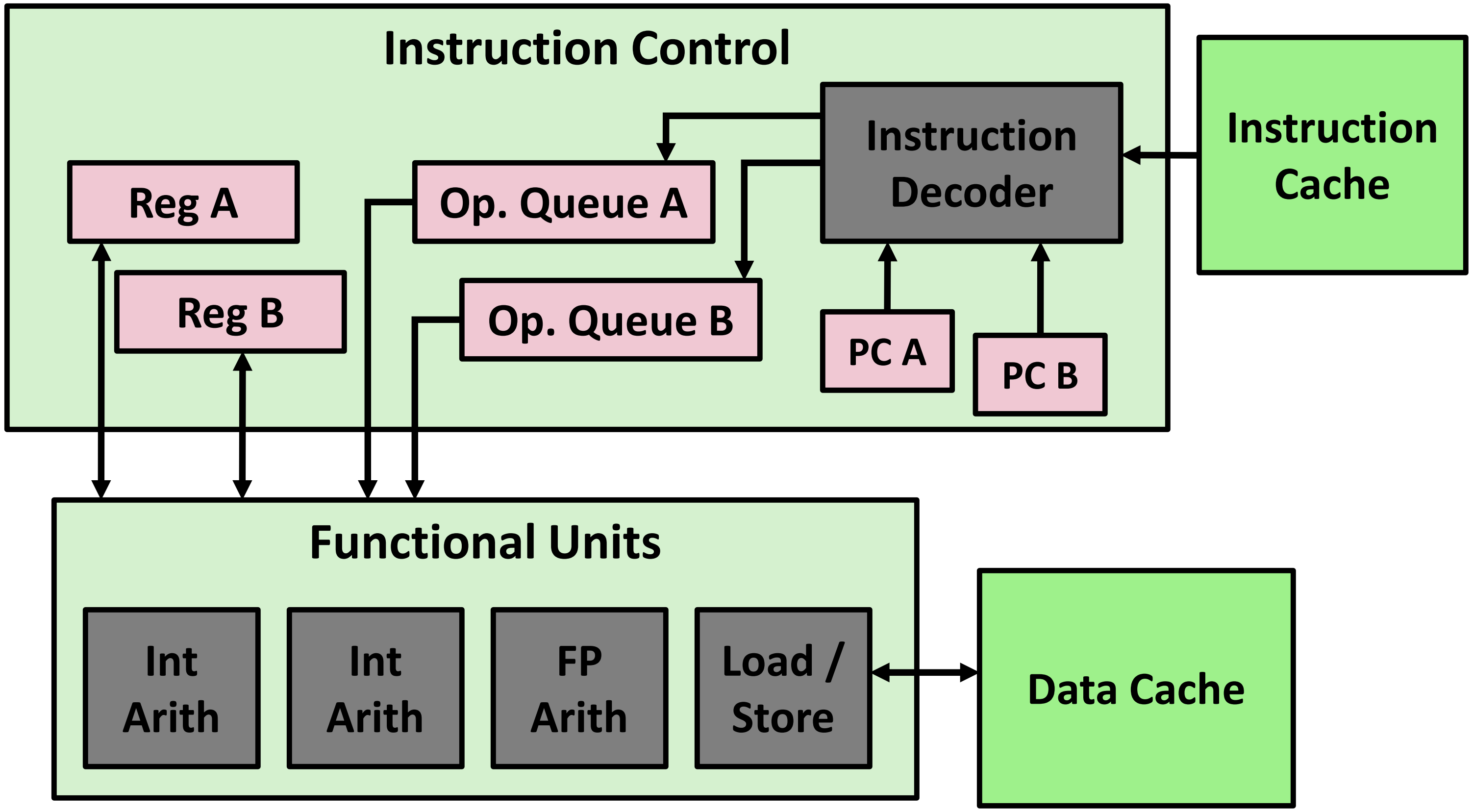


Out-of-Order Processor Structure



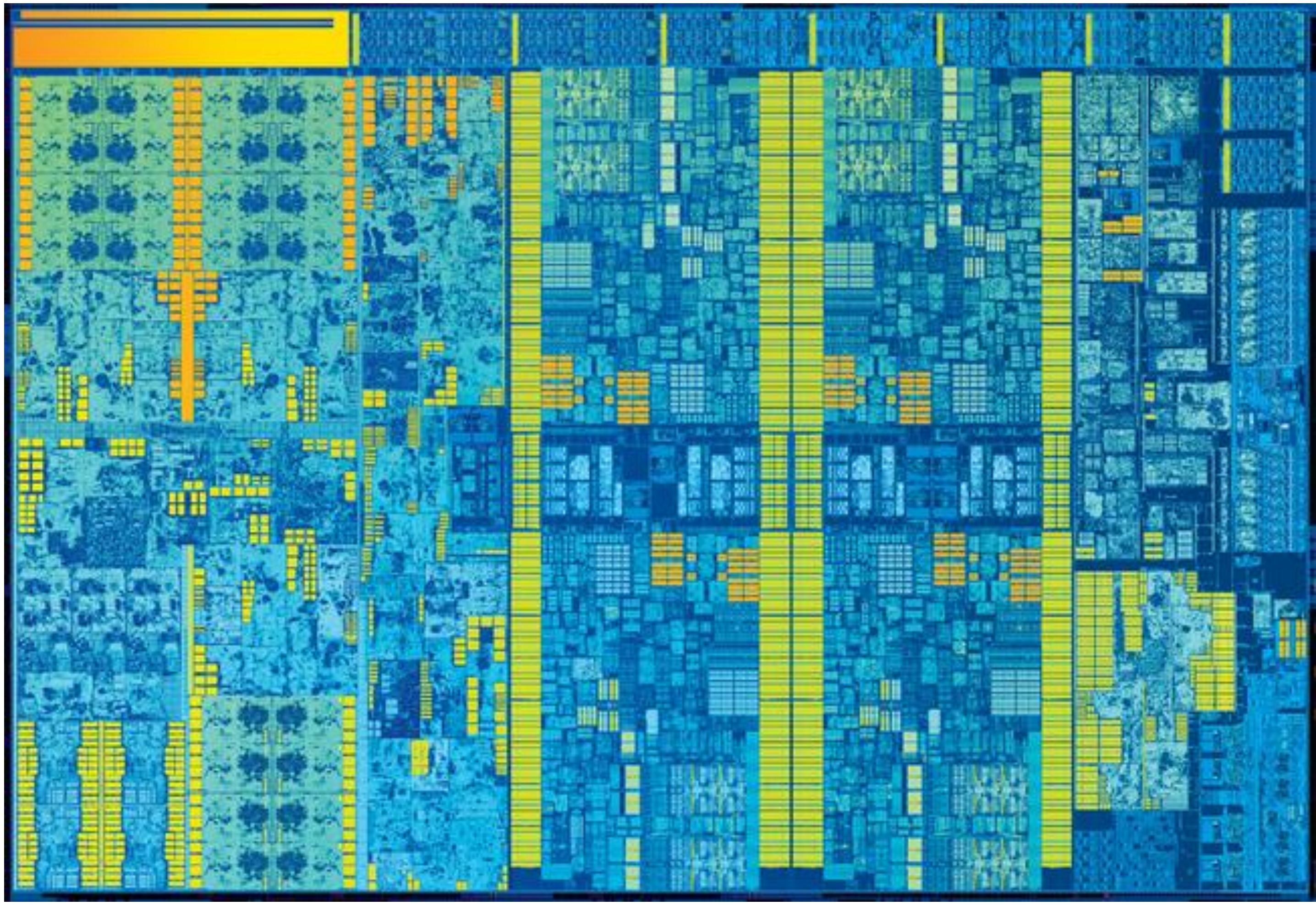
- Instruction control dynamically converts program into stream of operations
- Operations mapped onto functional units to execute in parallel

Hyperthreading Implementation

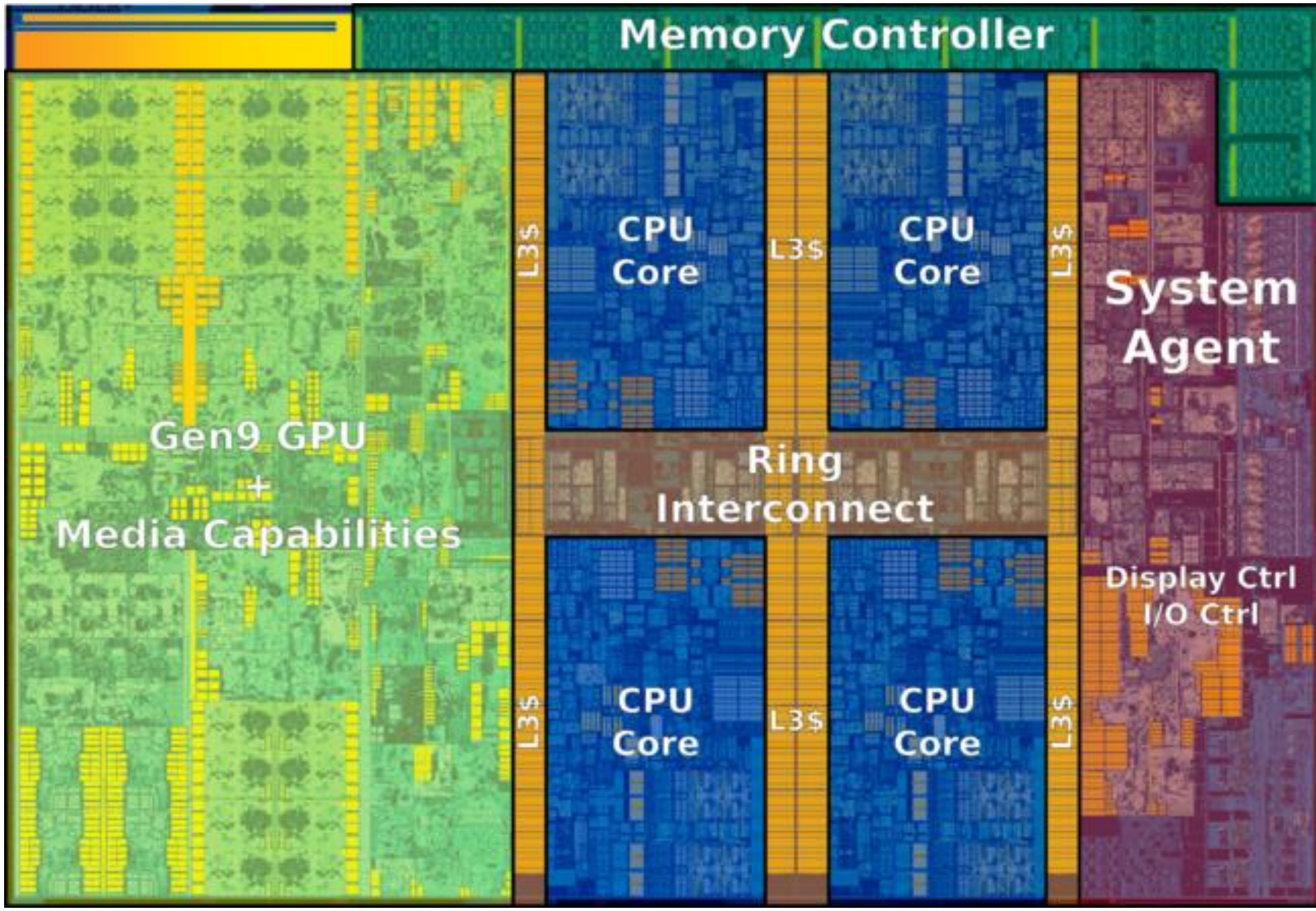


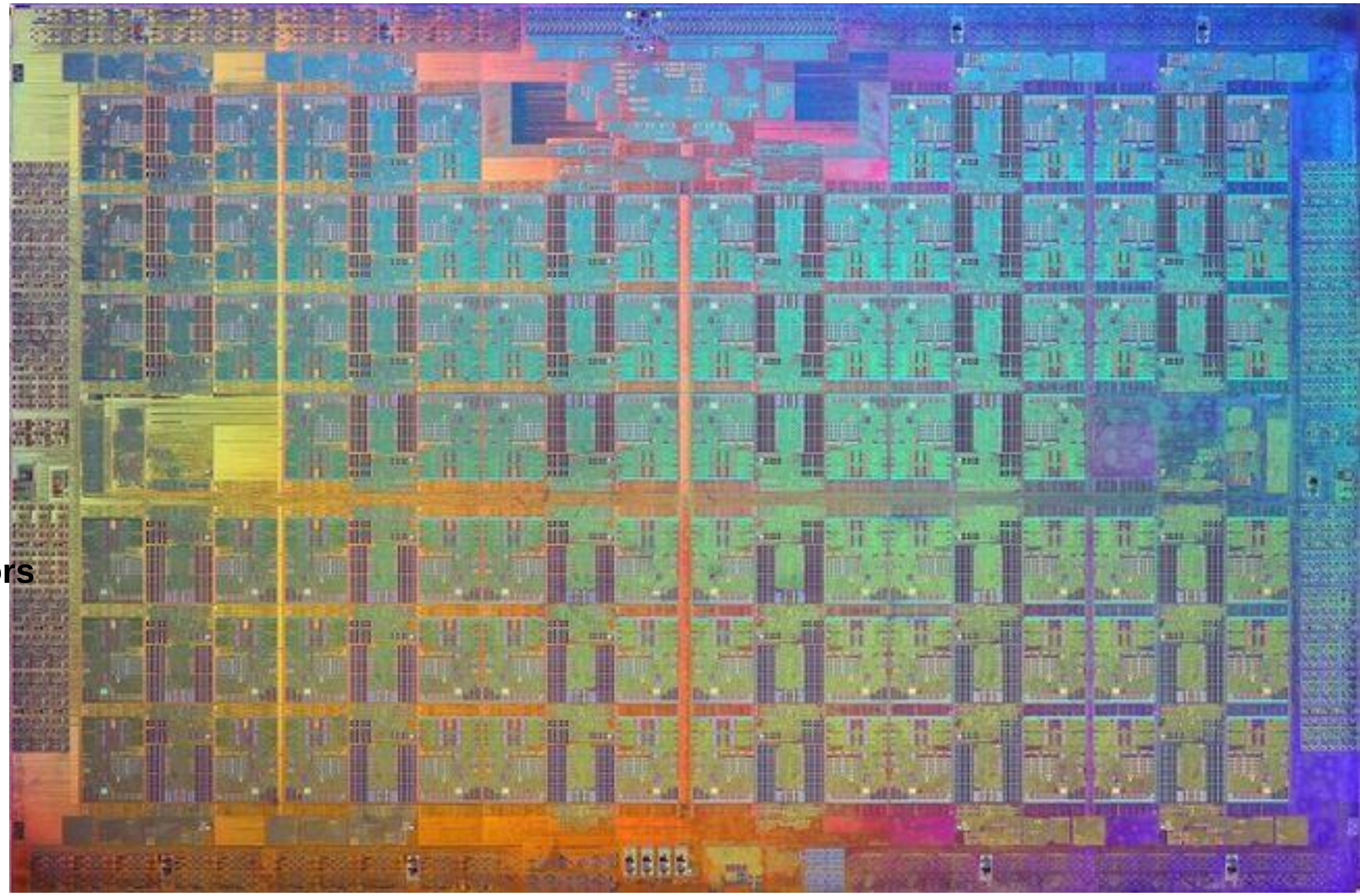
- Replicate instruction control to process K instruction streams
- K copies of all registers
- Share functional units

- Skylake
- 14 nm process
- 11 metal layers
- ~1,750,000,000 transistors
- ~9.19 mm x ~11.08 mm
- ~101.83 mm² die size
- 4 CPU cores + 24 GPU EUs



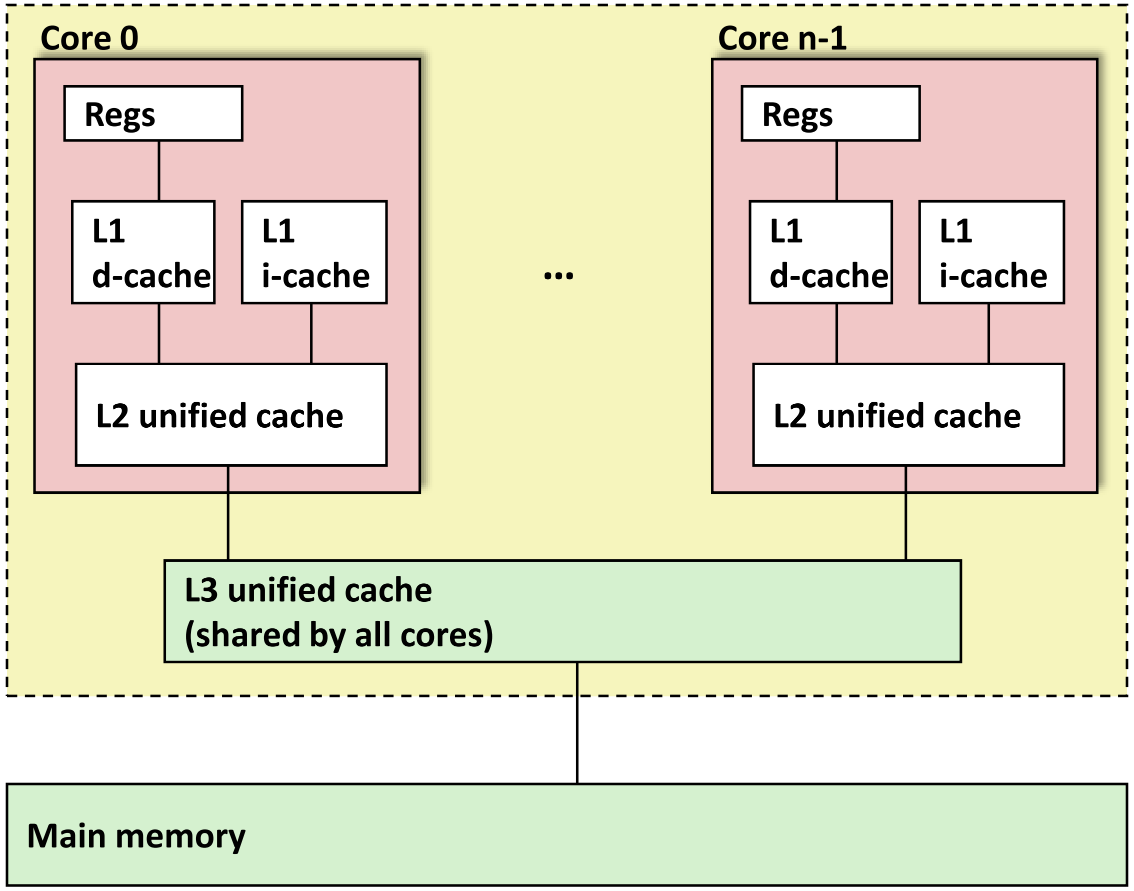
- 14 nm process
- 11 metal layers
- ~1,750,000,000 transistors
- ~9.19 mm x ~11.08 mm
- ~101.83 mm² die size
- 4 CPU cores + 24 GPU EUs

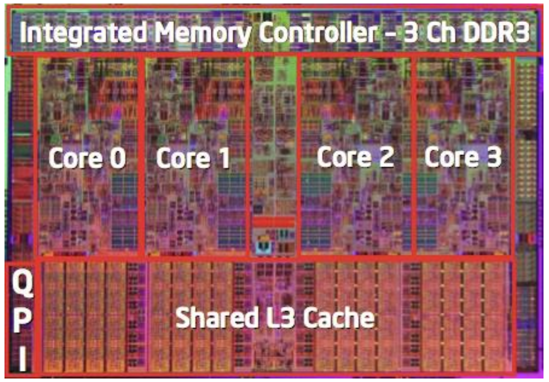




- <u>**Knight’s Landing**</u>
- <u>**14 nm process**</u>
- **682.6 mm² die size**
- **76 CPU cores**
- **7,100,000,000 transistors**
- Pairs of cores share L2$

Typical Multicore Processor





- Multiple processors operating with coherent view of memory

Benchmark Machine

- **Get data about machine from /proc/cpuinfo**
- **Shark Machines**
  - Intel Xeon E5520 @ 2.27 GHz
  - Nehalem, ca. 2010
  - 8 Cores
  - Each can do 2x hyperthreading

Exploiting parallel execution

- **So far, we've used threads to deal with I/O delays**
  - e.g., one thread per client to prevent one from delaying another
- **Multi-core CPUs offer another opportunity**
  - Spread work over threads executing in parallel on N cores
  - Happens automatically, if many independent tasks
    - e.g., running many applications or serving many clients
  - Can also write code to make one big task go faster
    - by organizing it as multiple parallel sub-tasks
- **Shark machines can execute 16 threads at once**
  - 8 cores, each with 2-way hyperthreading
  - Theoretical speedup of 16X
    - never achieved in our benchmarks

Today

- **Parallel Computing Hardware**
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core
- **Thread-Level Parallelism**
  - Splitting program into independent tasks
    - Example: Parallel summation
    - Amdahl's Law
- **Cache Coherence & Memory Consistency**
  - What happens when multiple threads are reading & writing shared state

Summation Example

- **Sum numbers 0, …, N-1**
  - Should add up to (N-1)*N/2
- **Partition into K ranges**
  - $\lfloor N/K \rfloor$ values each
  - Each of the *t* threads processes 1 range
  - Accumulate leftover values serially
- **Method #1: All threads update single global variable**
  - 1A: No synchronization
  - 1B: Synchronize with pthread semaphore
  - 1C: Synchronize with pthread mutex
    - "Binary" semaphore. Only values 0 & 1

Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;
```

Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;
```

Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;

/* Number of elements summed by each thread */
size_t nelems_per_thread;

/* Keep track of thread IDs */
pthread_t tid[MAXTHREADS];

/* Identify each thread */
int myid[MAXTHREADS];
```

Accumulating in Single Global Variable: Operation

```
nelems_per_thread = nelems / nthreads;

/* Set global value */
global_sum = 0;

/* Create threads and wait for them to finish */
for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
}
for (i = 0; i < nthreads; i++)
    Pthread_join(tid[i], NULL);

result = global_sum;

/* Add leftover elements */
for (e = nthreads * nelems_per_thread; e < nelems; e++)
     result += e;
```

Thread ID

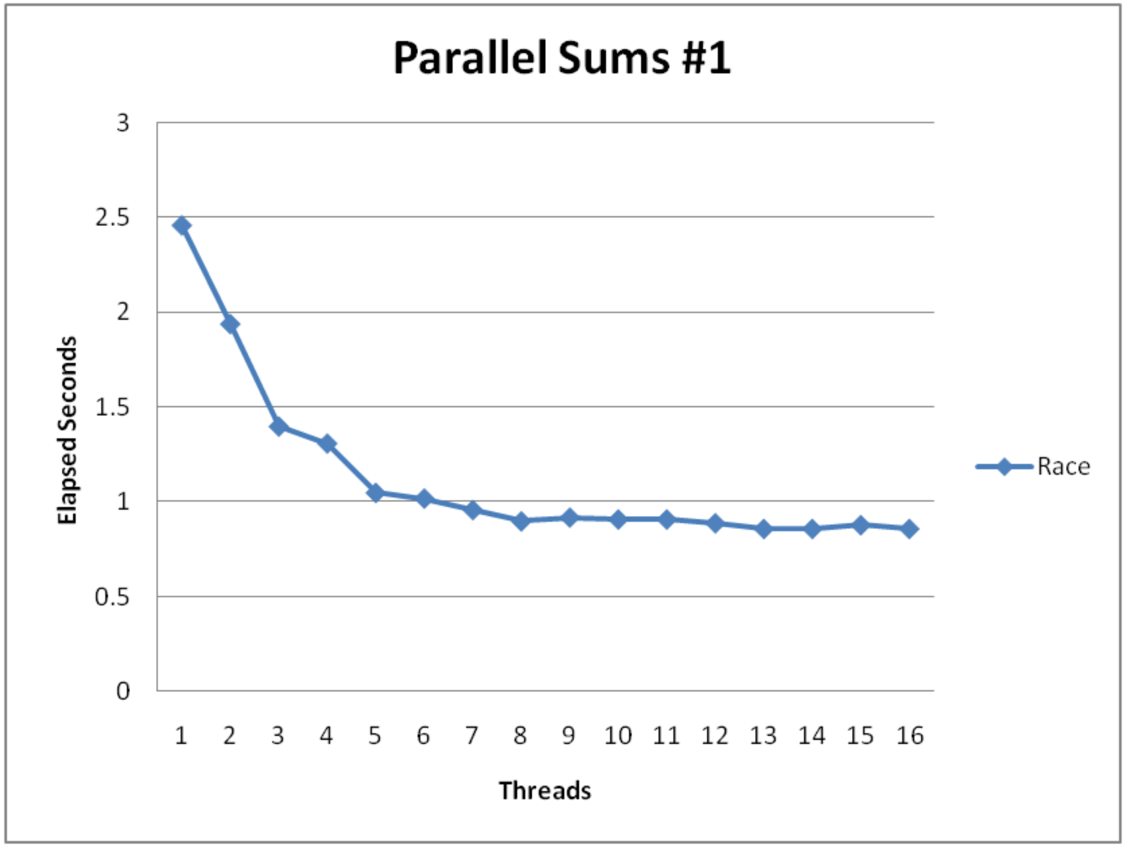
Thread routine

*Thread arguments (void *p)*

Thread Function: No Synchronization

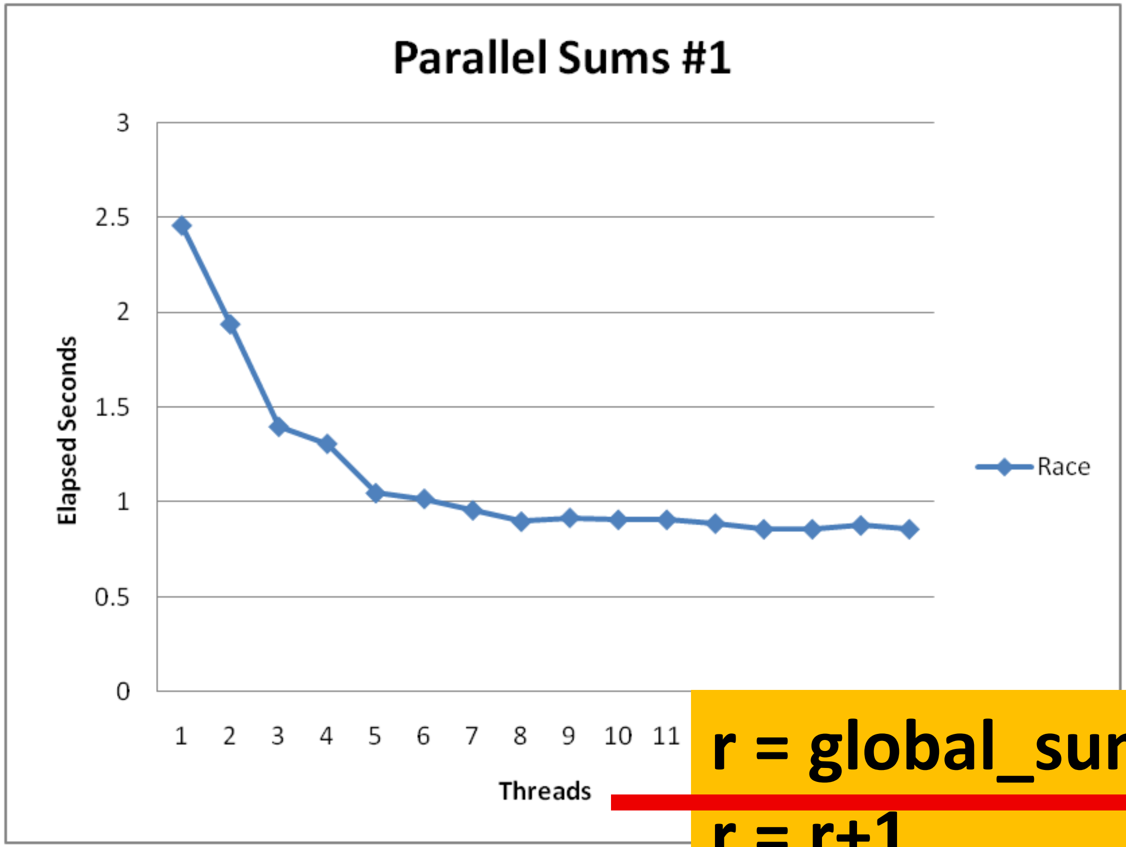
```
void *sum_race(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        global_sum += i;
    }
    return NULL;
}
```

Unsynchronized Performance



- N = 2^{30}
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread! Why?

Unsynchronized Performance



```
r = global_sum
r = r+1
global_sum = r
```

- $N = 2^{30}$
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread! Why?

Thread Function: Semaphore / Mutex

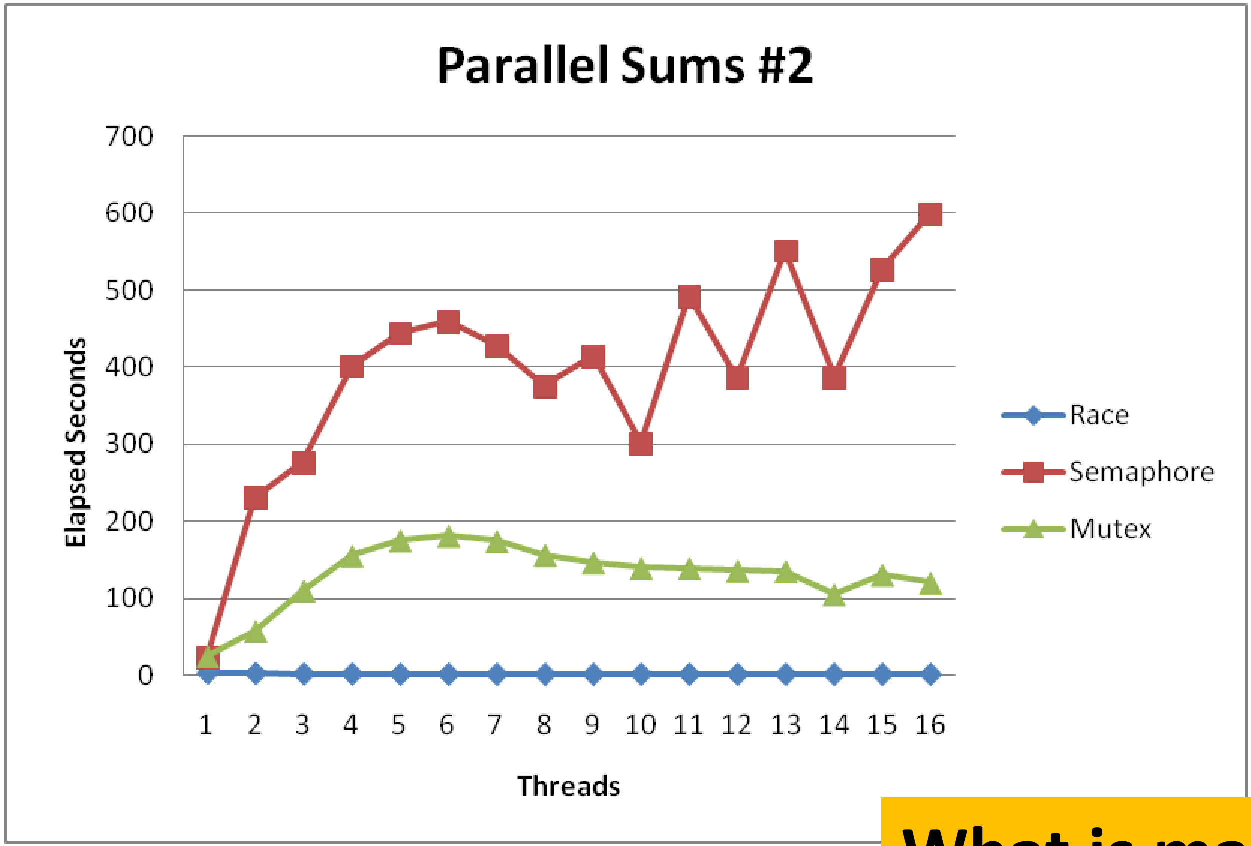
Semaphore

```
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

Mutex

```
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

Semaphore / Mutex Performance



- **Terrible Performance**
  - 2.5 seconds ➔ ~10 minutes
- **Mutex 3X faster than semaphore**
- **Clearly, neither is successful**

What is main reason for poor performance?

Separate Accumulation

- **Method #2: Each thread accumulates into separate variable**
  - 2A: Accumulate in contiguous array elements
  - 2B: Accumulate in spaced-apart array elements
  - 2C: Accumulate in registers

```
/* Partial sum computed by each thread */
data_t psum[MAXTHREADS*MAXSPACING];

/* Spacing between accumulators */
size_t spacing = 1;
```

Separate Accumulation: Operation

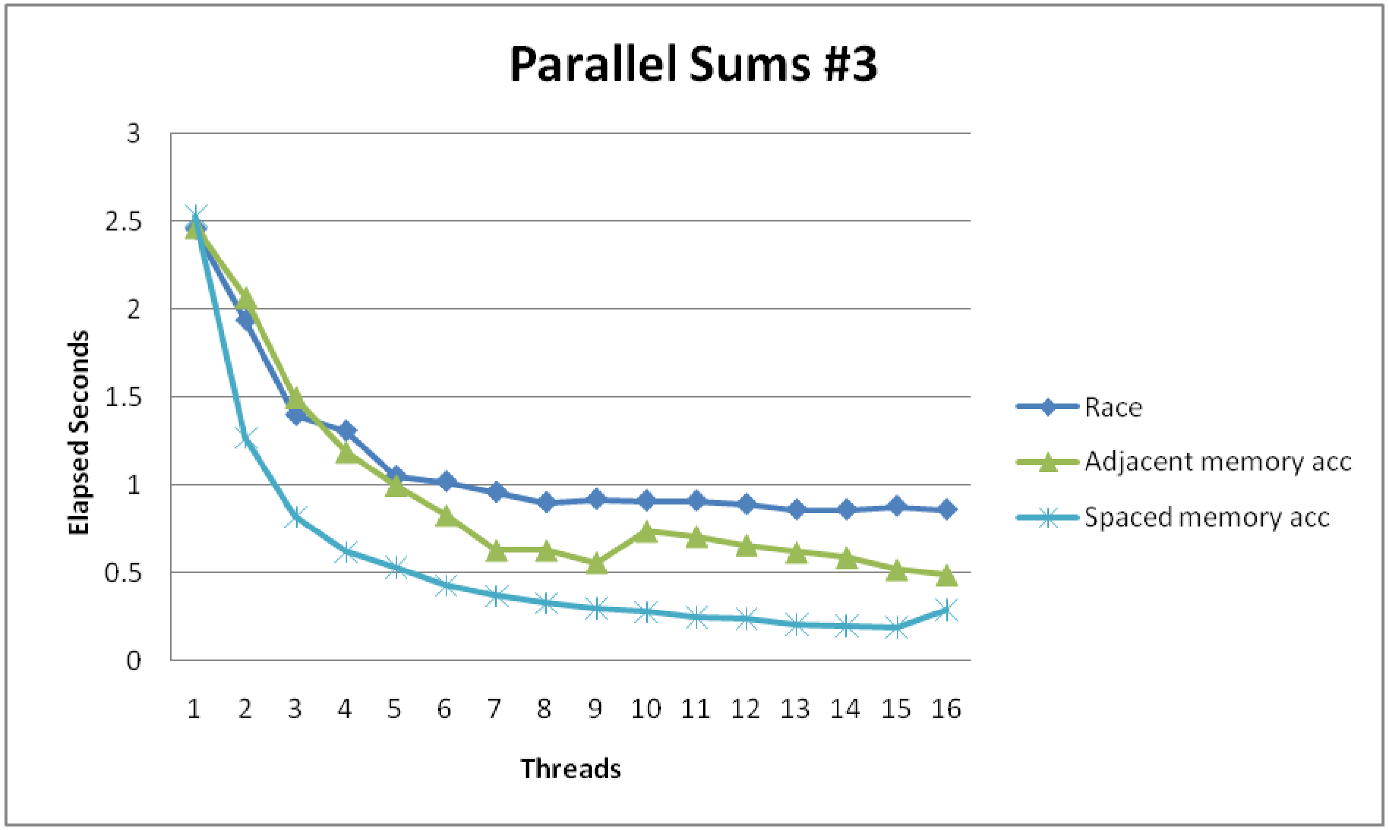
```
    nelems_per_thread = nelems / nthreads;

    /* Create threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        psum[i*spacing] = 0;
        Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    result = 0;

    /* Add up the partial sums computed by each thread */
    for (i = 0; i < nthreads; i++)
        result += psum[i*spacing];

    /* Add leftover elements */
    for (e = nthreads * nelems_per_thread; e < nelems; e++)
         result += e;
```

Thread Function: Memory Accumulation

Where is the mutex?

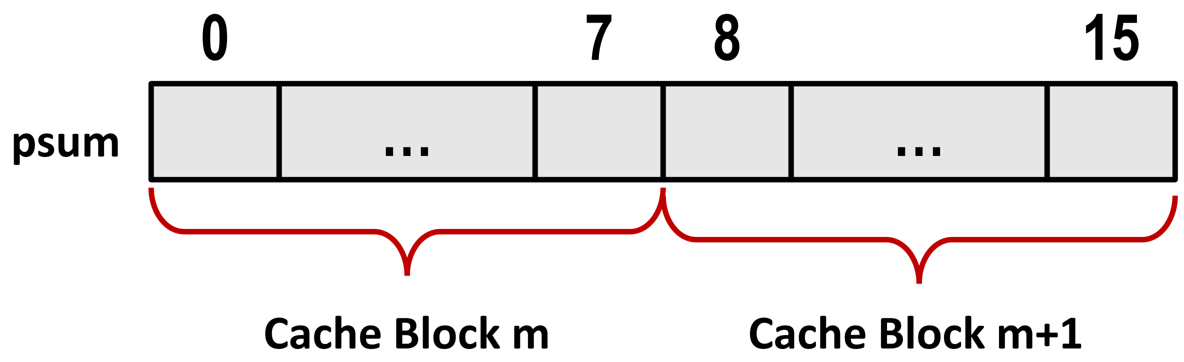
```
void *sum_global(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    size_t index = myid*spacing;
    psum[index] = 0;
    for (i = start; i < end; i++) {
        psum[index] += i;
    }
    return NULL;
}
```

Memory Accumulation Performance



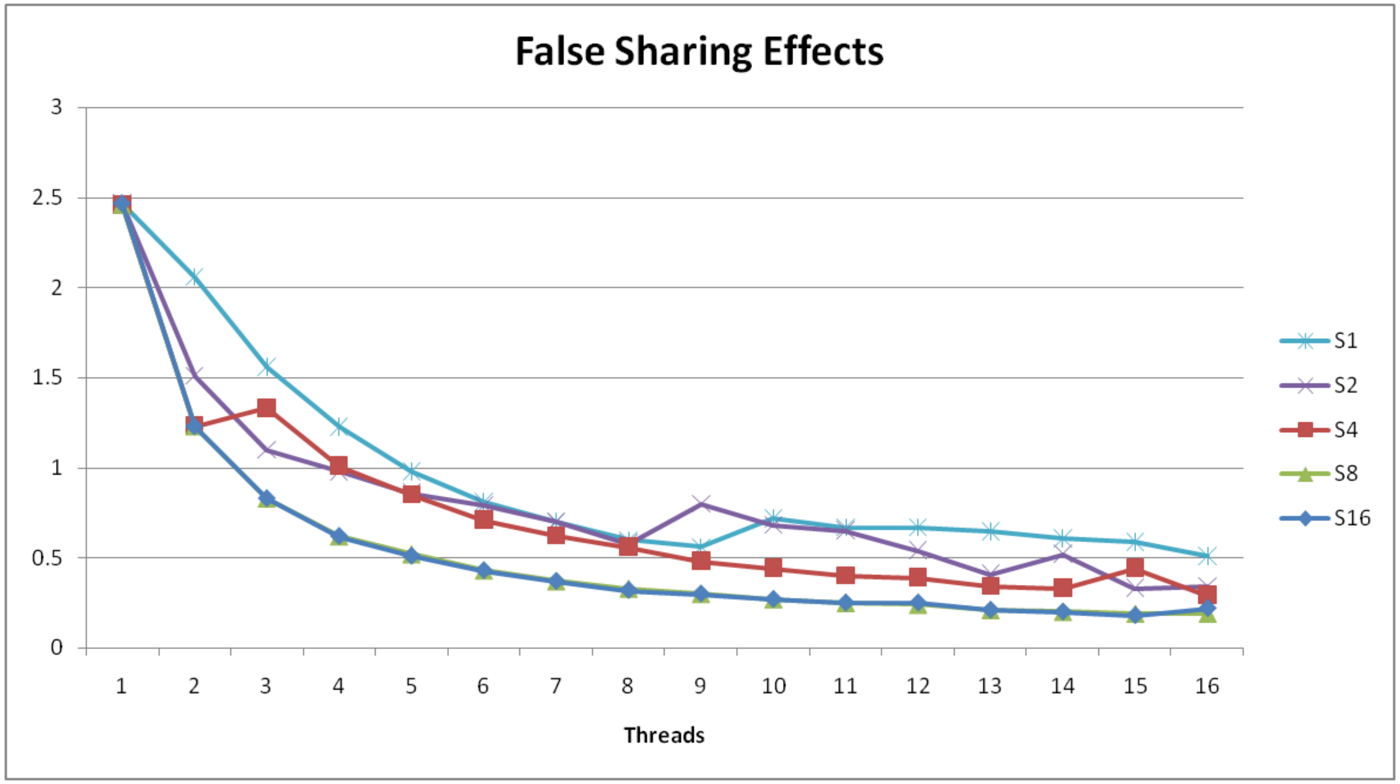
- **Clear threading advantage**
  - Adjacent speedup: 5 X
  - Spaced-apart speedup: 13.3 X (Only observed speedup > 8)
- **Why does spacing the accumulators apart matter?**

False Sharing



- **Coherency maintained on cache blocks**
- **To update psum[i], thread i must have exclusive access**
  - Threads sharing common cache block will keep fighting each other for access to block

False Sharing Performance

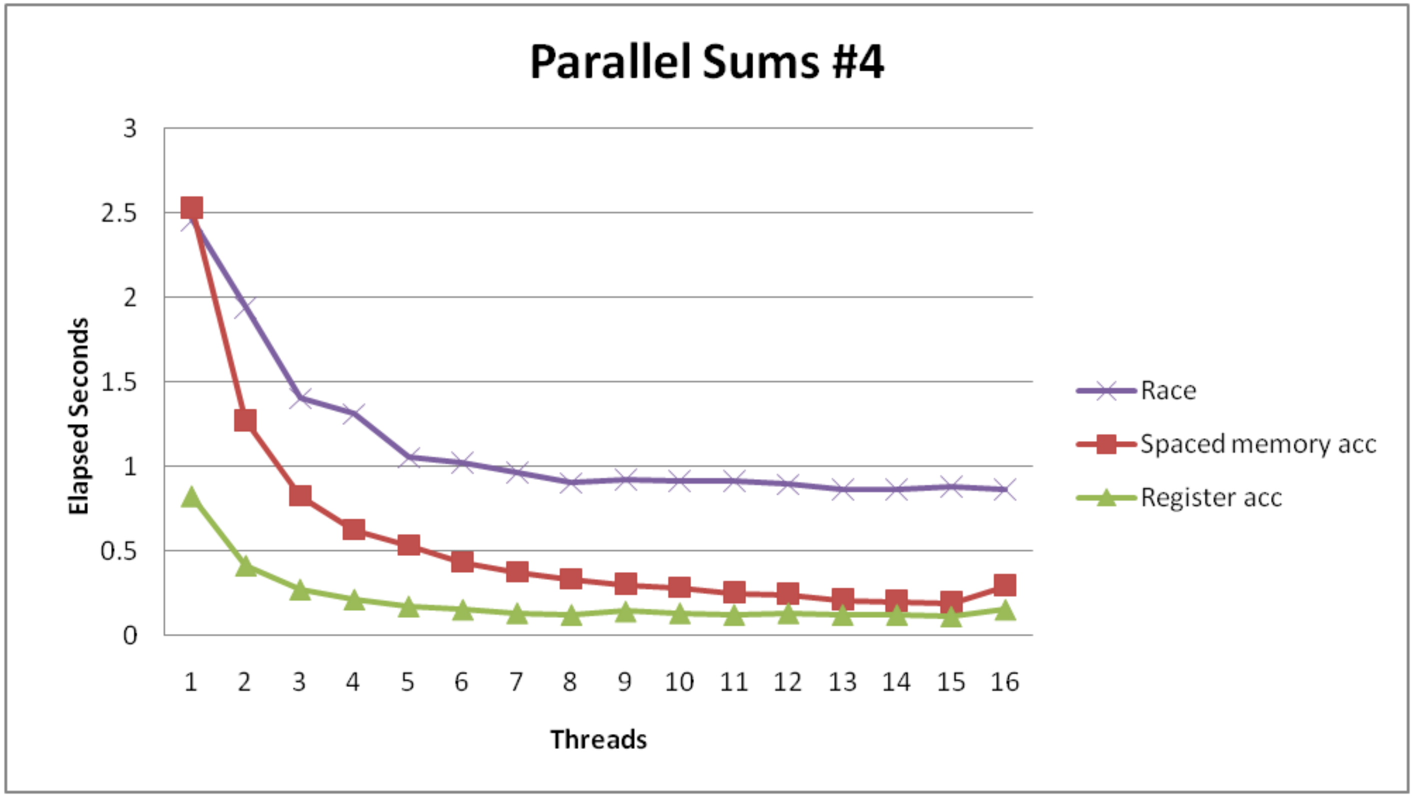


  - Best spaced-apart performance 2.8 X better than best adjacent
- **Demonstrates cache block size = 64**
  - 8-byte values
  - No benefit increasing spacing beyond 8

Thread Function: Register Accumulation

```
void *sum_local(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    size_t index = myid*spacing;
    data_t sum = 0;
    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[index] = sum;
    return NULL;
}
```

Register Accumulation Performance



- **Clear threading advantage**
  - Speedup = 7.5 X vs. shared accumulator
- **2X better than fastest separate memory accumulation**

Lessons learned

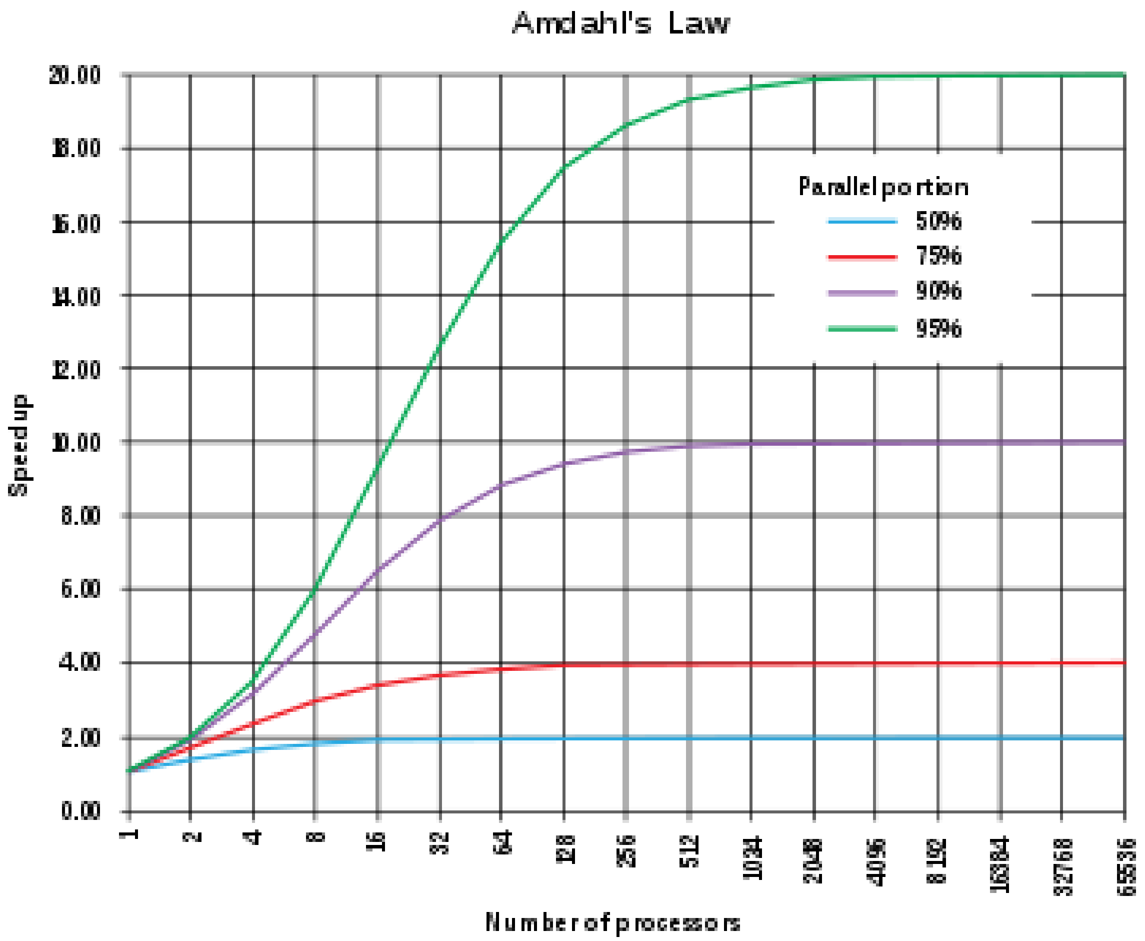
- **Sharing memory can be expensive**
  - Pay attention to true sharing
  - Pay attention to false sharing
- **Use registers whenever possible**
  - (Remember cachelab)
  - Use local cache whenever possible
- **Deal with leftovers**
- **When examining performance, compare to best possible sequential implementation**

Amdahl's Law

- **Overall problem**
  - T Total sequential time required
  - p Fraction of total that can be sped up ($0 \leq p \leq 1$)
  - k Speedup factor
- **Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - Portion which can be sped up runs k times faster
    - Portion which cannot be sped up stays the same
  - Maximum possible speedup
    - $k = \infty$
    - $T_\infty = (1-p)T$

Amdahl's Law Example

- **Overall problem**
  - T = 10 Total time required
  - p = 0.9 Fraction of total which can be sped up
  - k = 9 Speedup factor
- **Resulting Performance**
  - $T_9 = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$
  - Maximum possible speedup
    - $T_\infty = 0.1 * 10.0 = 1.0$



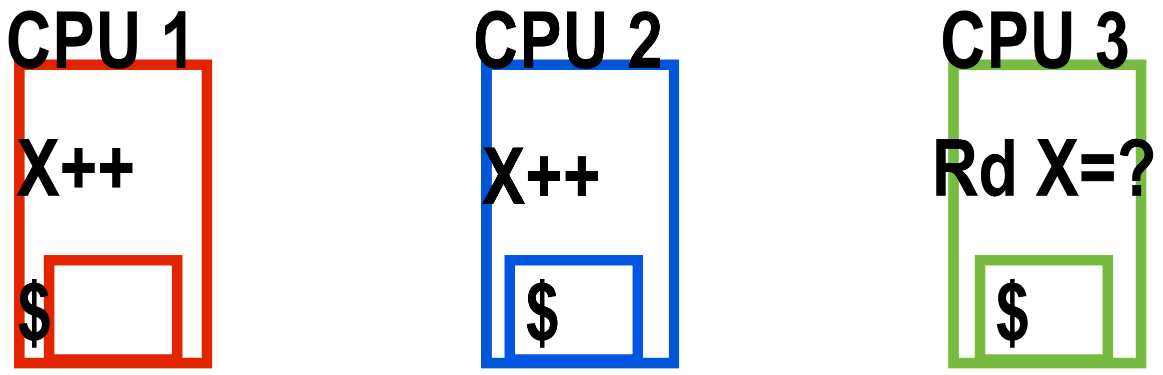
Amdahl's Corollary: Parallel speedup is limited by fraction of the program that is **parallelizable**

Quiz Time!

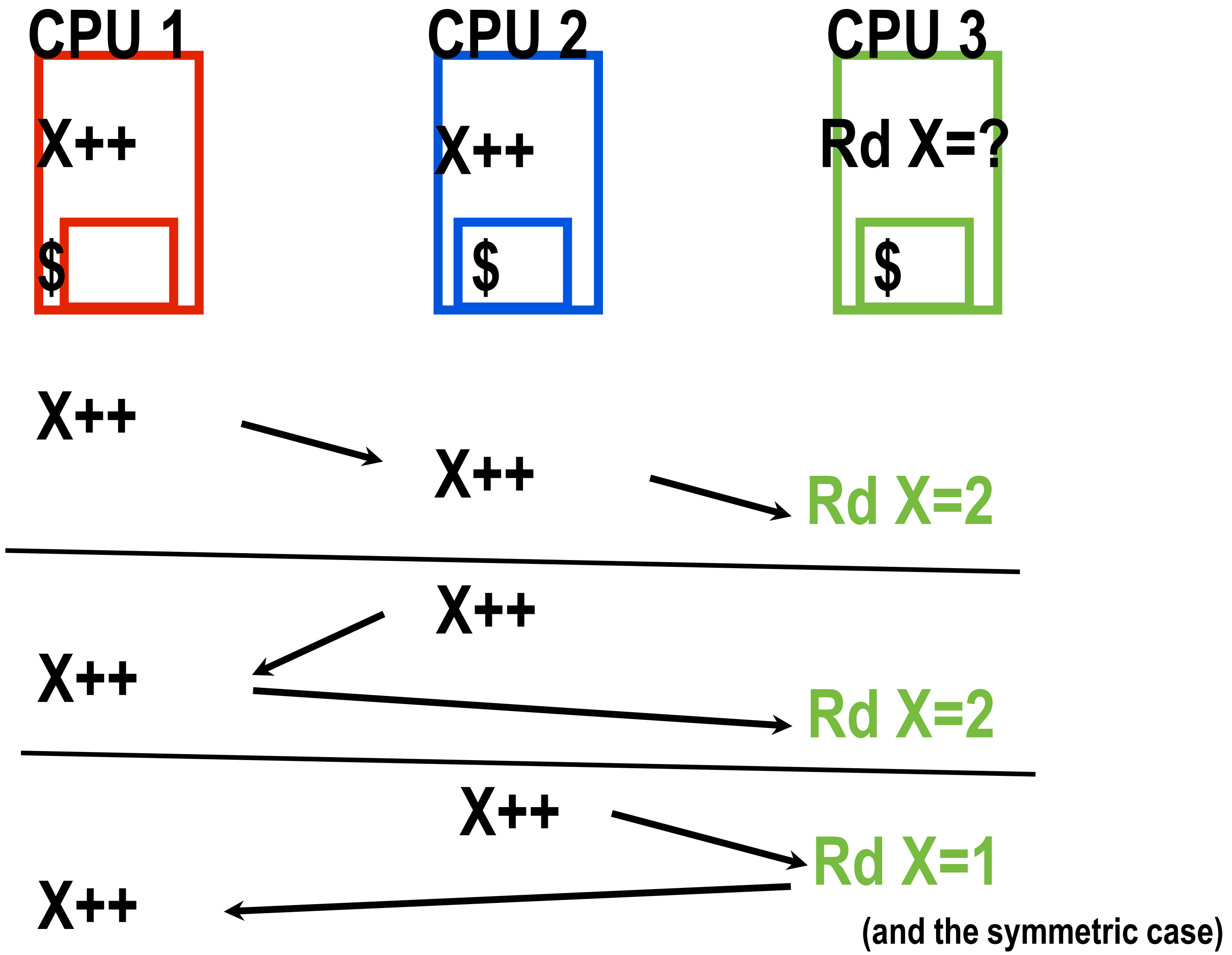
Check out:

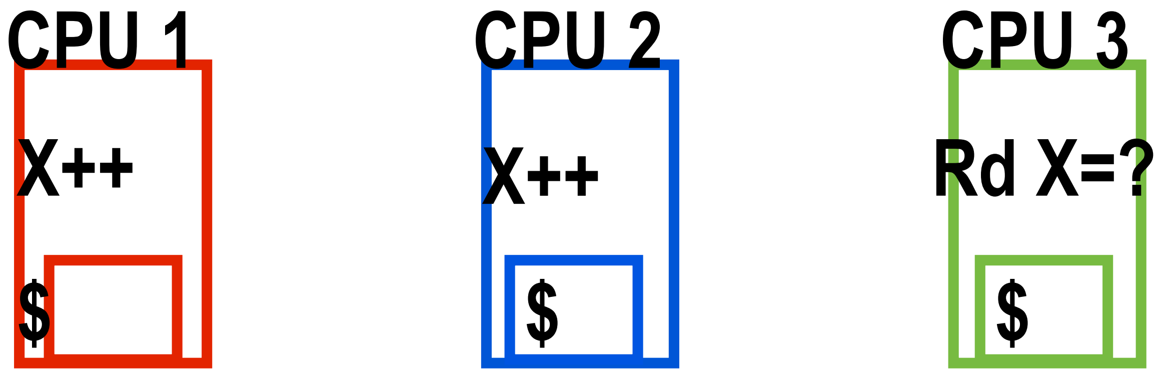
https://canvas.cmu.edu/courses/8555

Cache Coherence

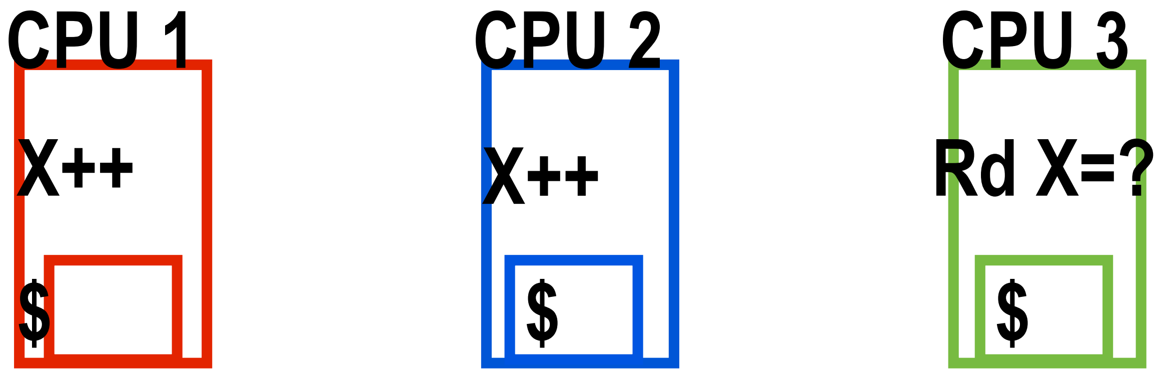


What is the behavior of this parallel program? (X initially 0)

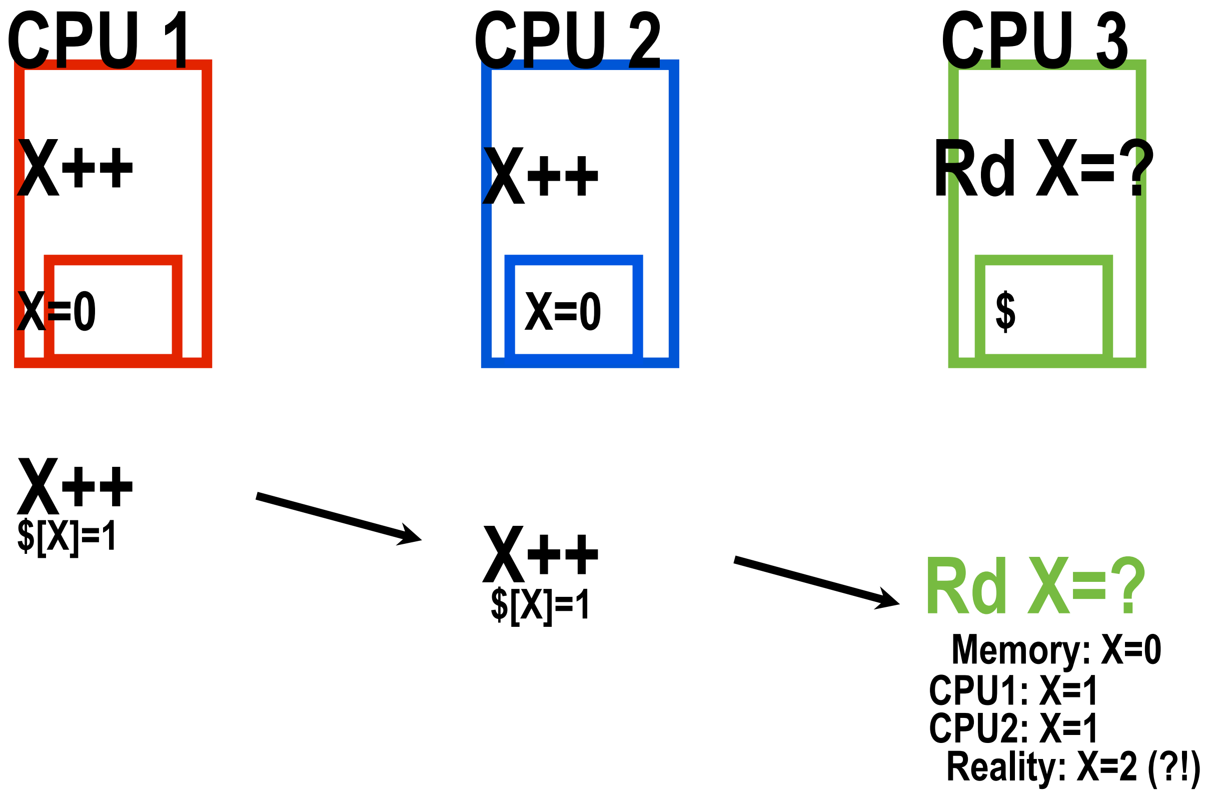

CPU 1
X++
$
CPU 2
X++
$
CPU 3
Rd X=?
$
X++
X++
Rd X=2
X++
X++
Rd X=2
X++
Rd X=1
X++
(and the symmetric case)



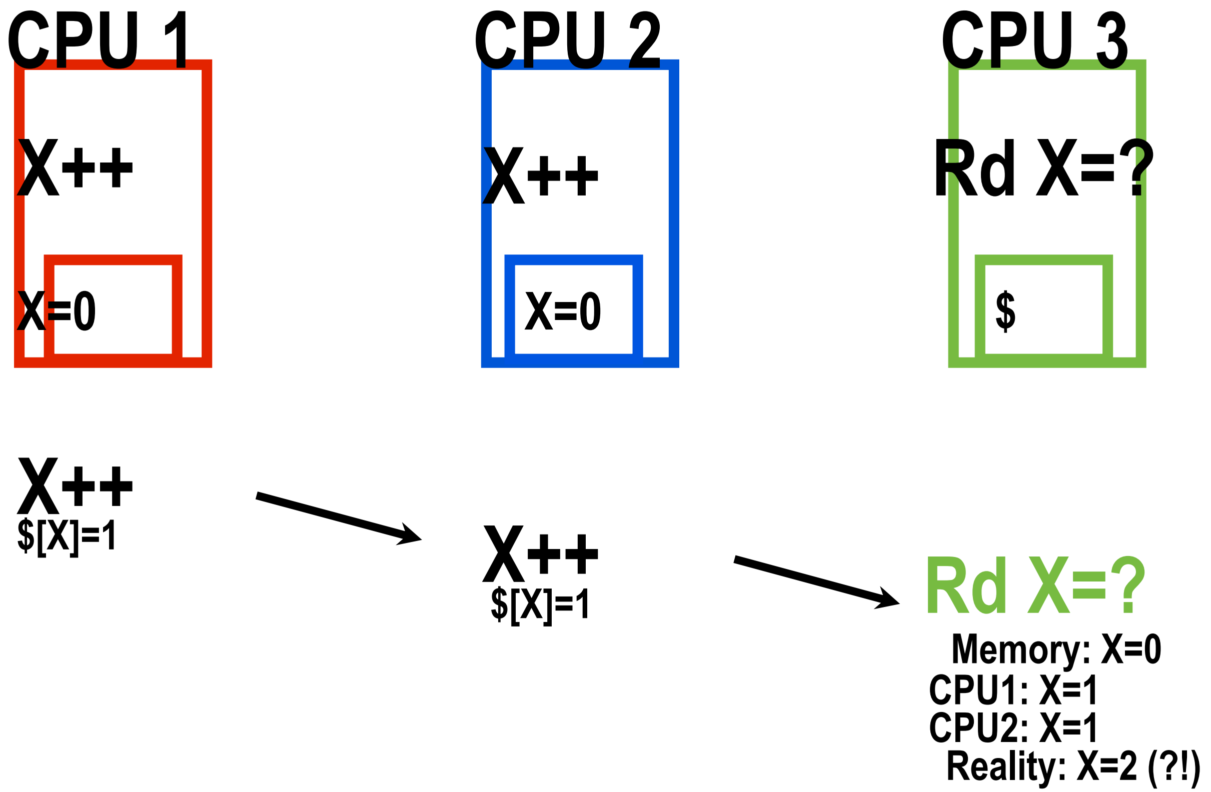
What assumptions are we making about the system to produce the results 0, 1, and 2?



We assume the updates see one anothers' results!
(Why wouldn't they?)



So what the heck do we do now?



Never let this happen. Caches should be coherent.

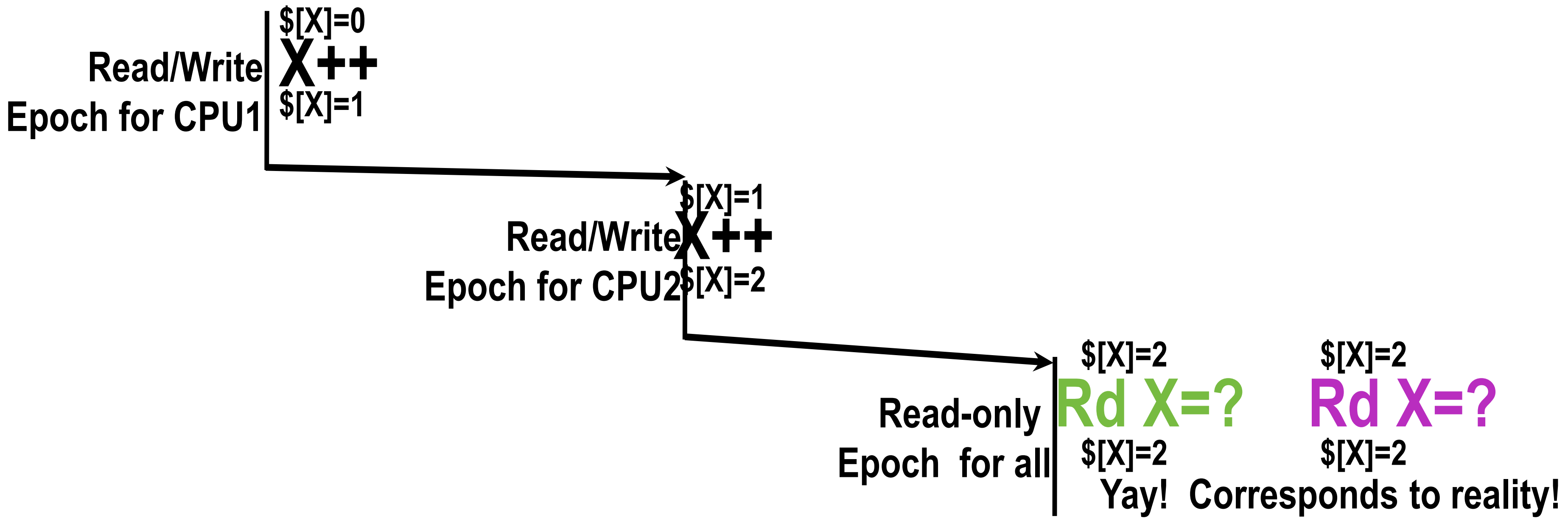
"coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores"

Informally Defining Coherence

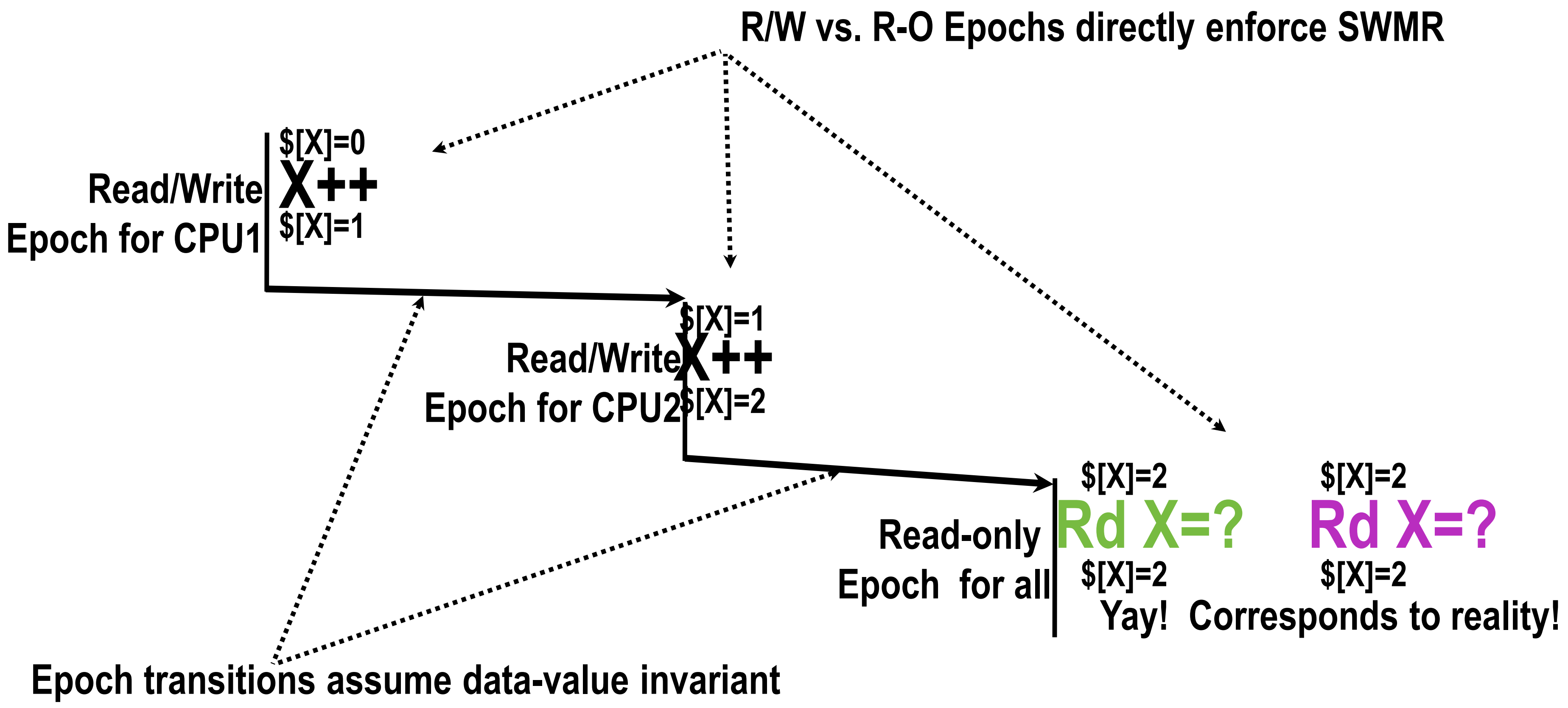
"Coherence serializes all reads with all updates to the same location by different CPUs/caches, so that each read sees the result of the most recent update by any other"

"Single Writer/Multiple Reader (SWMR) Invariant
+
Data-Value Invariant"

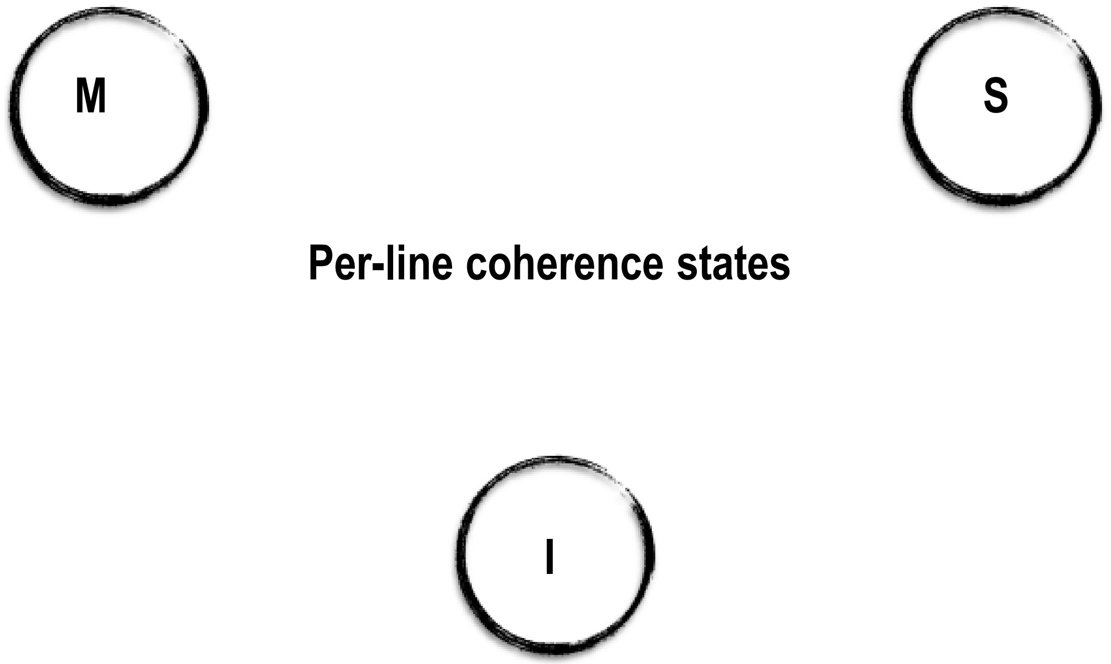
Epoch Model


$[X]=0
Read/Write Epoch for CPU1
X++
$[X]=1
$[X]=1
Read/Write Epoch for CPU2
X++
$[X]=2
$[X]=2
$[X]=2
Read-only Epoch for all
Rd X=?
Rd X=?
$[X]=2
$[X]=2
Yay! Corresponds to reality!

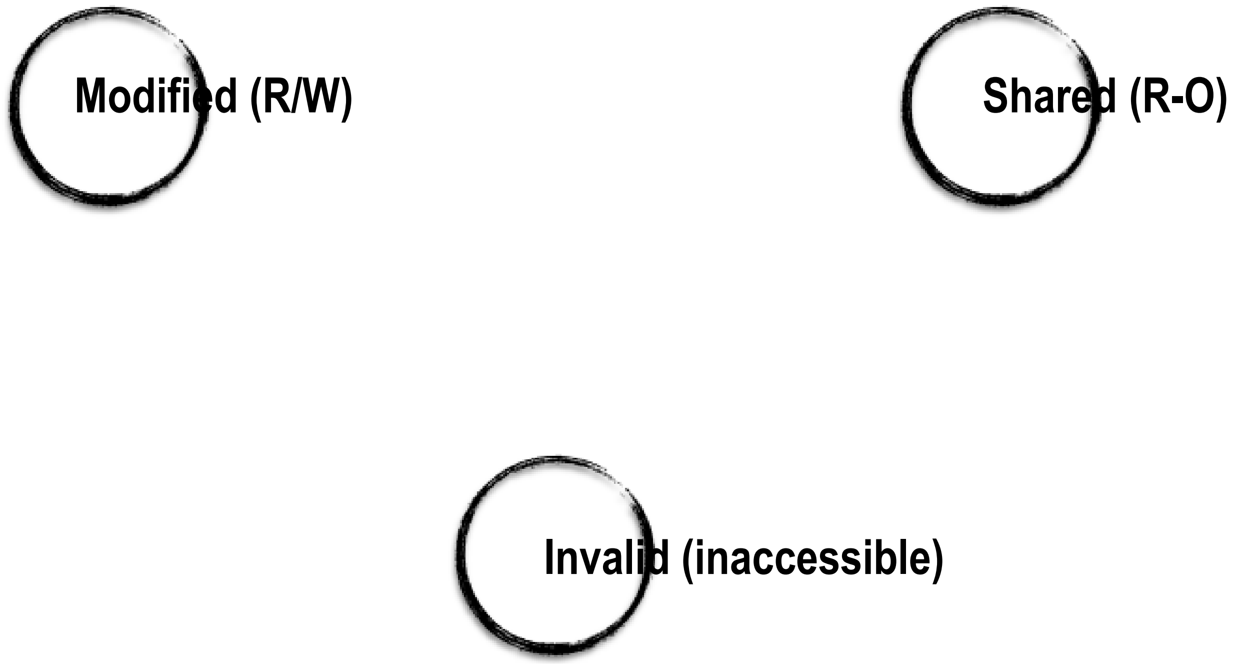
Epoch Model


R/W vs. R-O Epochs directly enforce SWMR
$[X]=0
Read/Write Epoch for CPU1
X++
$[X]=1
$[X]=1
Read/Write Epoch for CPU2
X++
$[X]=2
Read-only Epoch for all
$[X]=2
Rd X=?
$[X]=2
$[X]=2
Rd X=?
$[X]=2
Yay! Corresponds to reality!
Epoch transitions assume data-value invariant

Cache Coherence Protocol

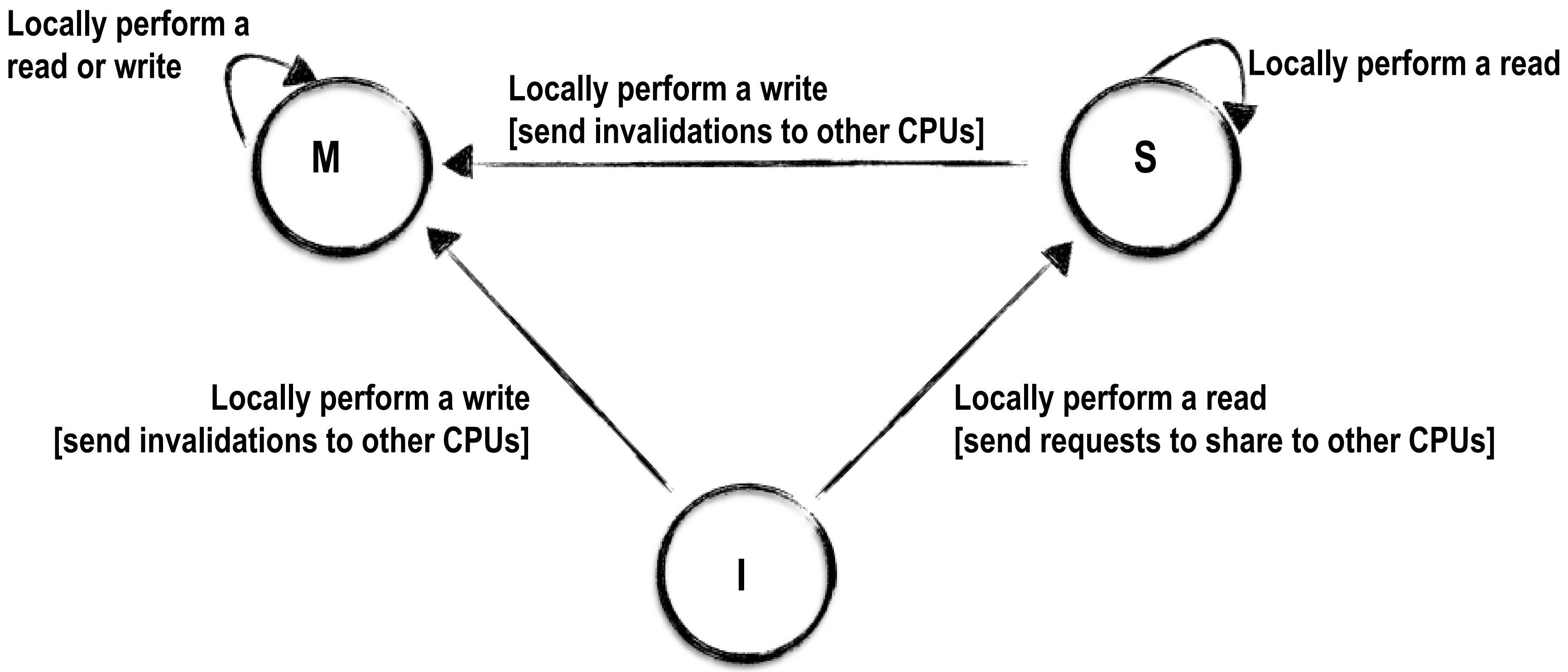


Cache Coherence Protocol



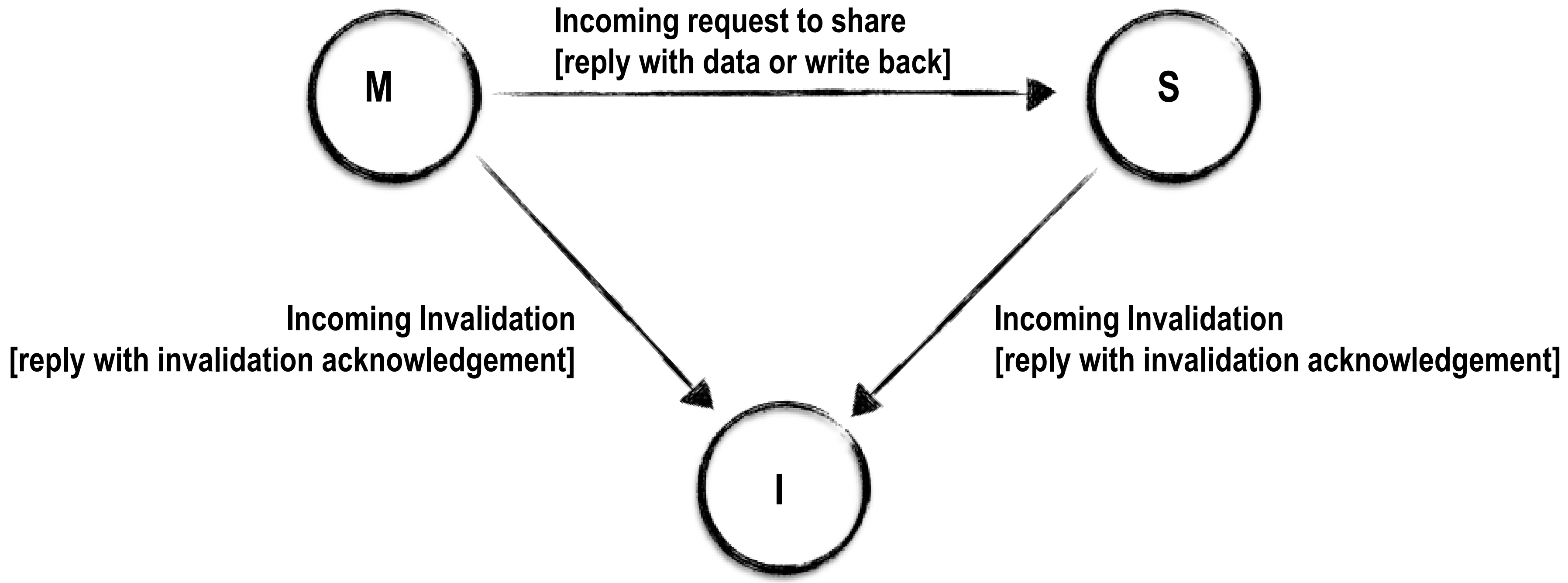
Cache Coherence Protocol

Local operations perspective

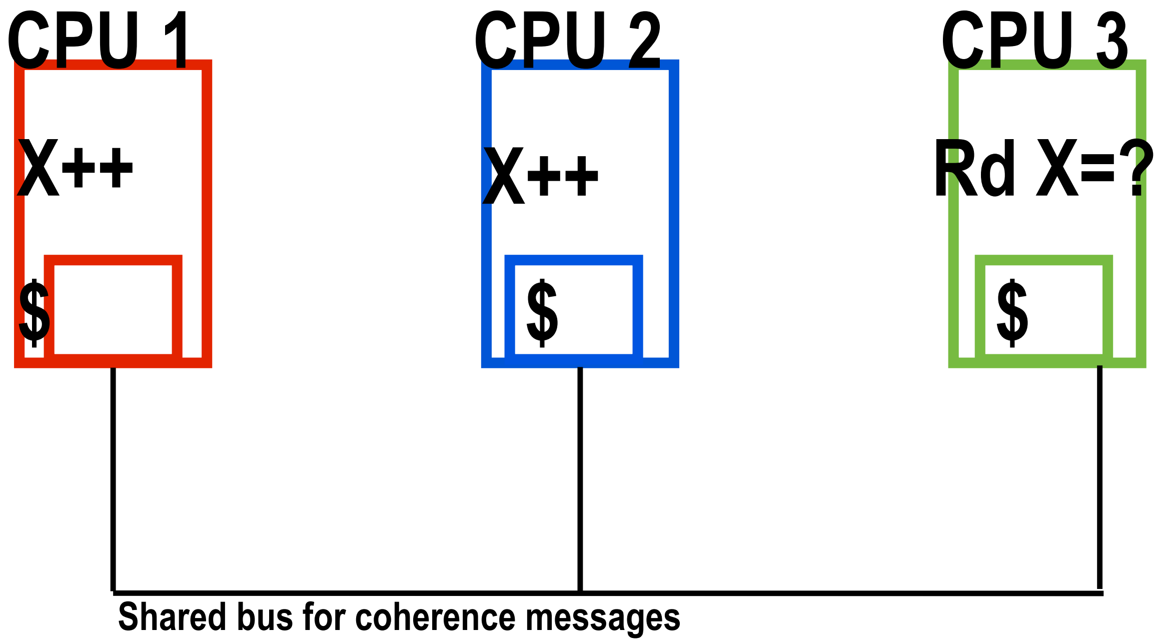

Locally perform a read or write
Locally perform a write [send invalidations to other CPUs]
Locally perform a read
M
S
Locally perform a write [send invalidations to other CPUs]
Locally perform a read [send requests to share to other CPUs]
I

Cache Coherence Protocol

Remote operations perspective

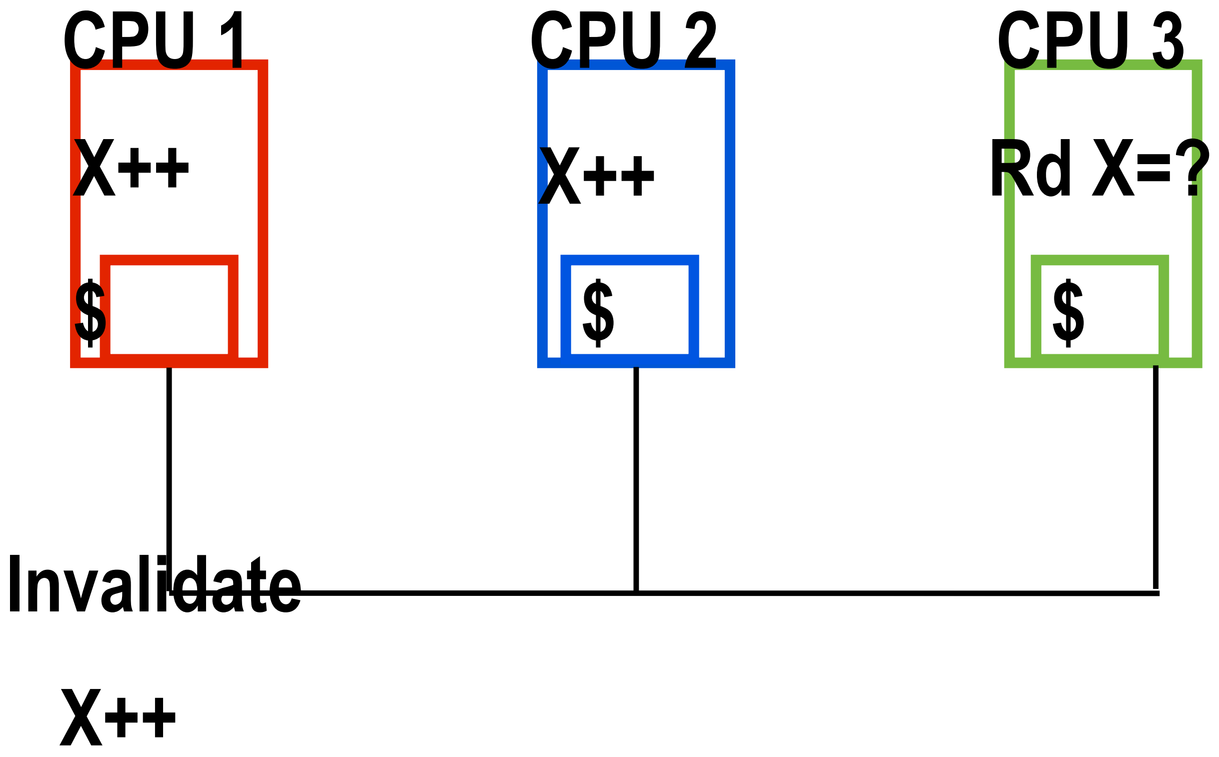

M
S
I
Incoming request to share
[reply with data or write back]
Incoming Invalidation
[reply with invalidation acknowledgement]
Incoming Invalidation
[reply with invalidation acknowledgement]

Implementing the Protocol

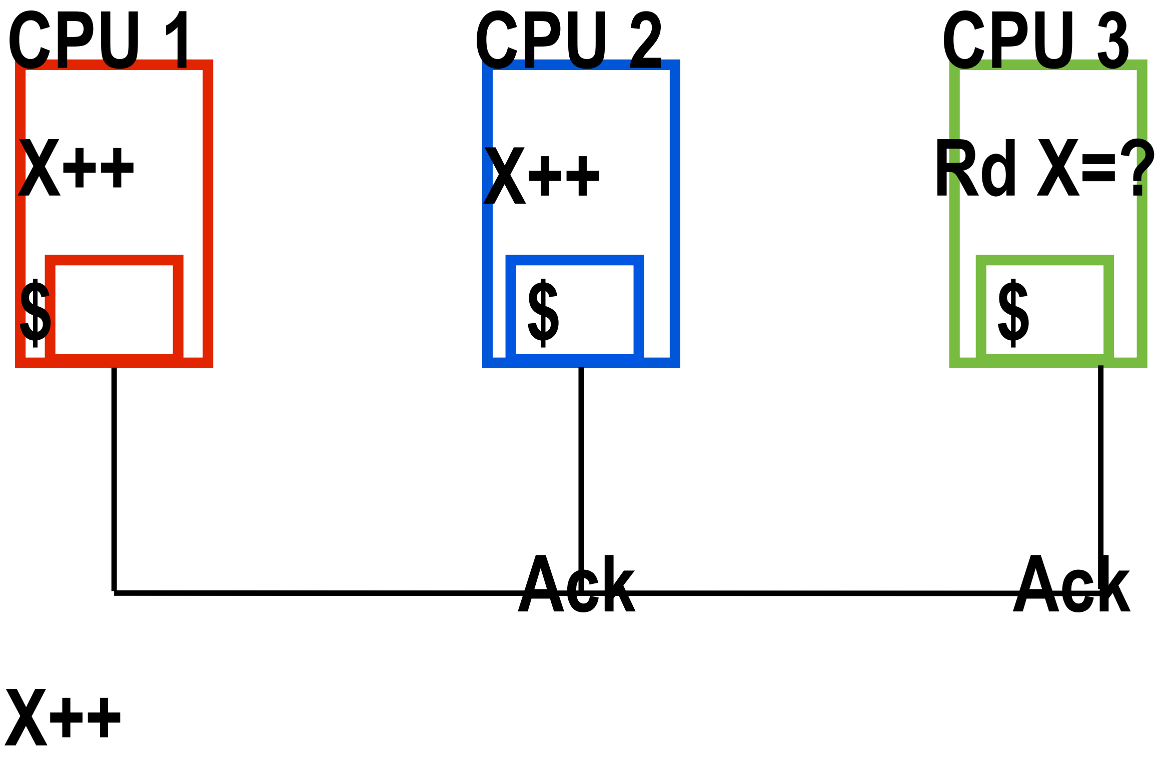


Snoopy Coherence

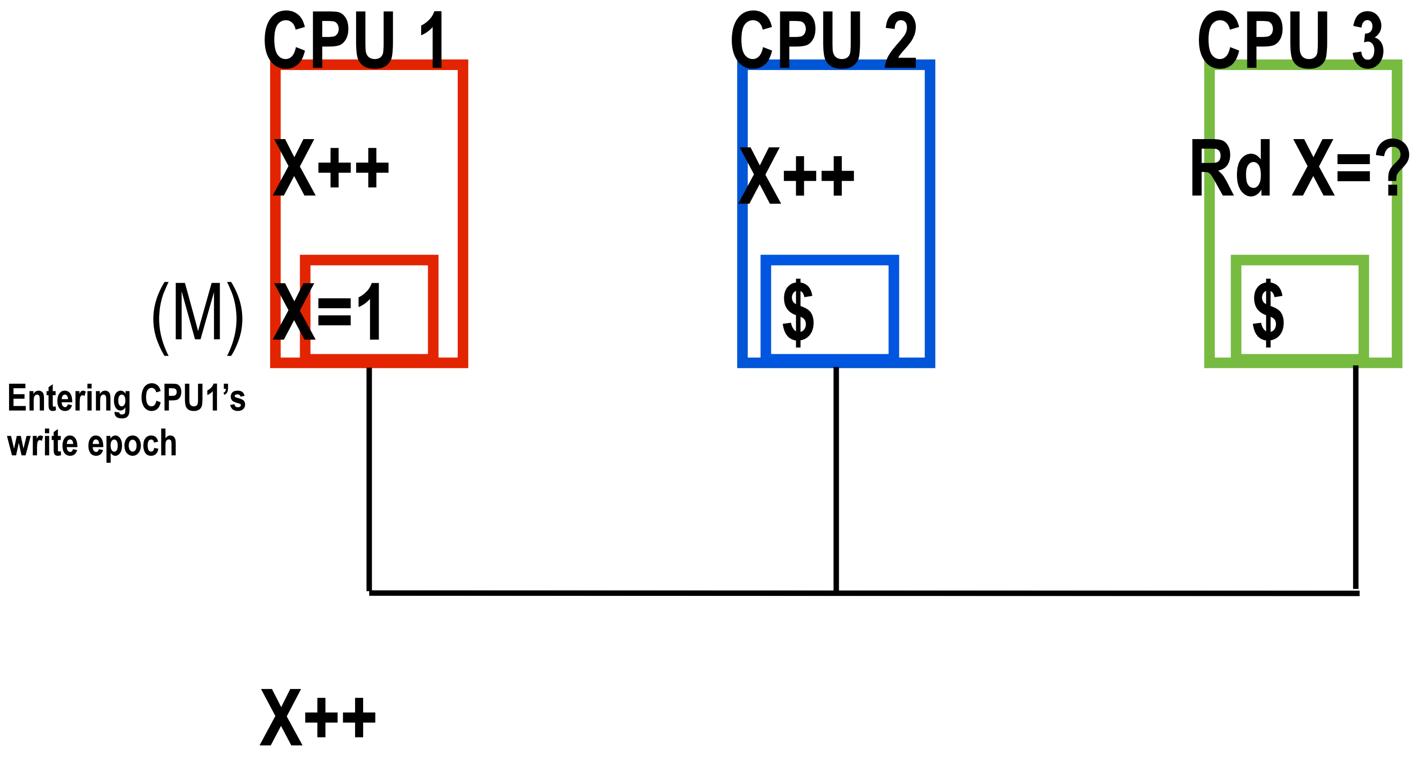
Implementing the Protocol



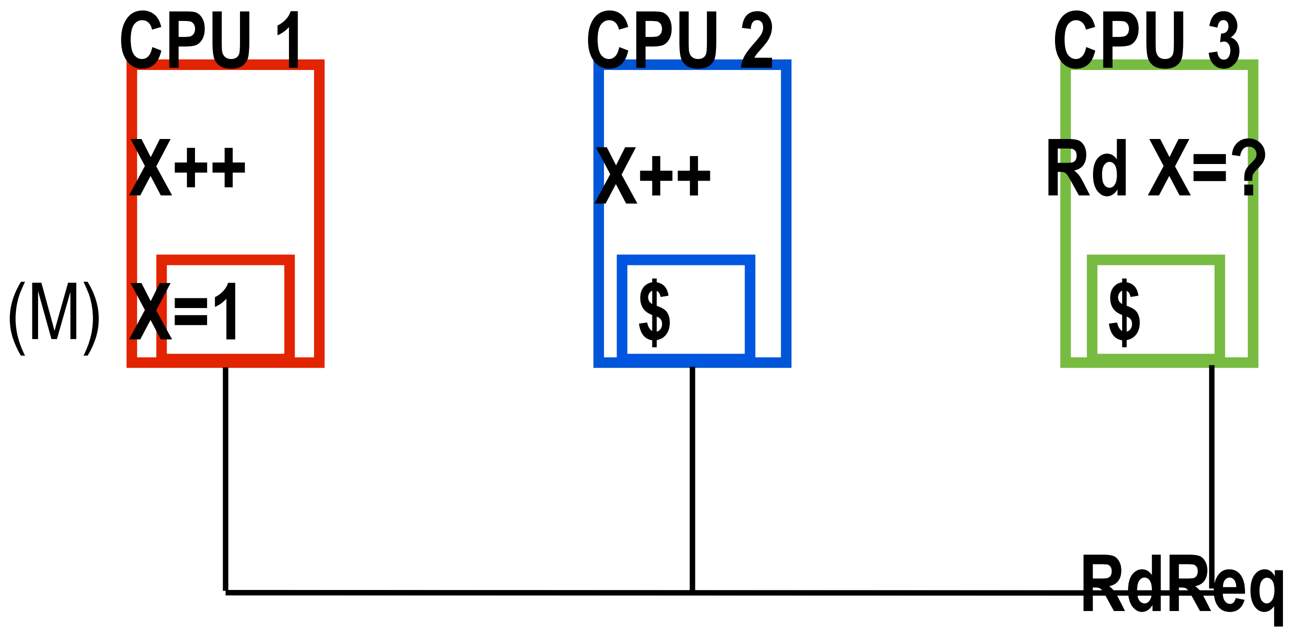
Implementing the Protocol



Implementing the Protocol



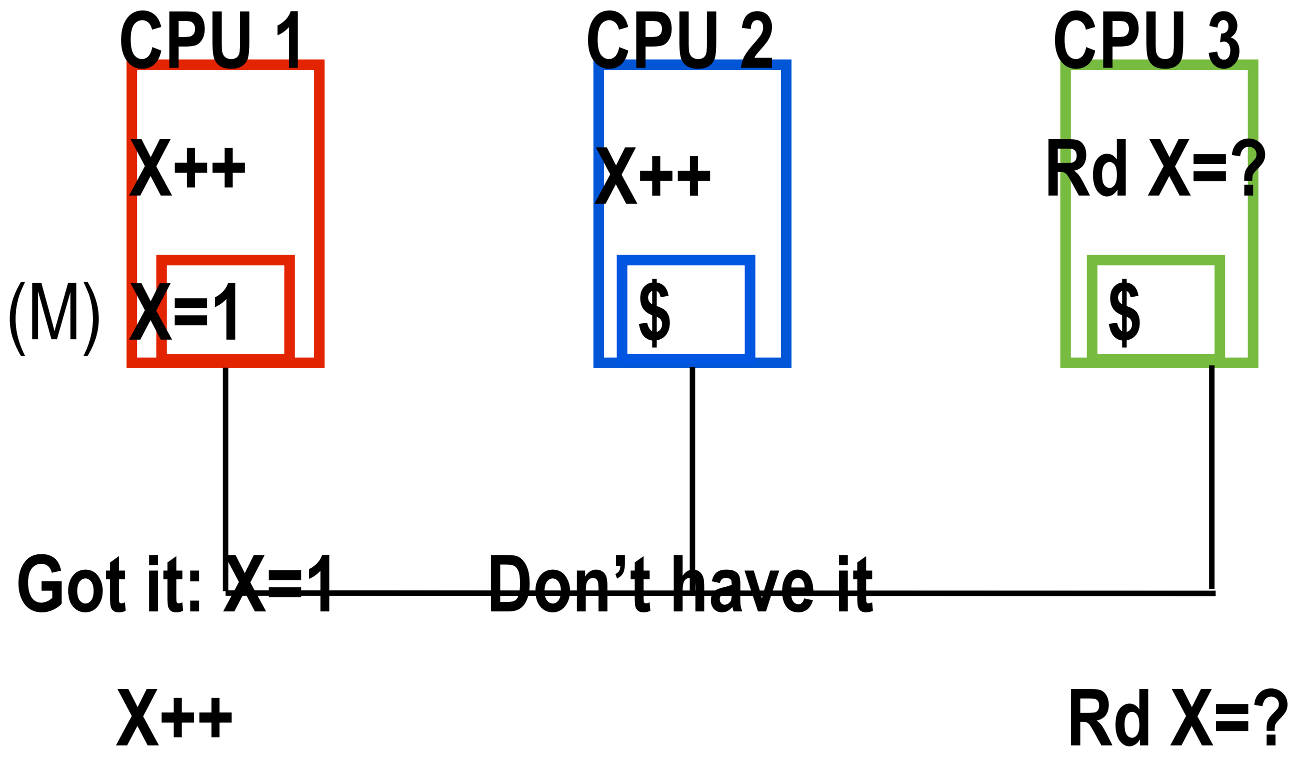
Implementing the Protocol



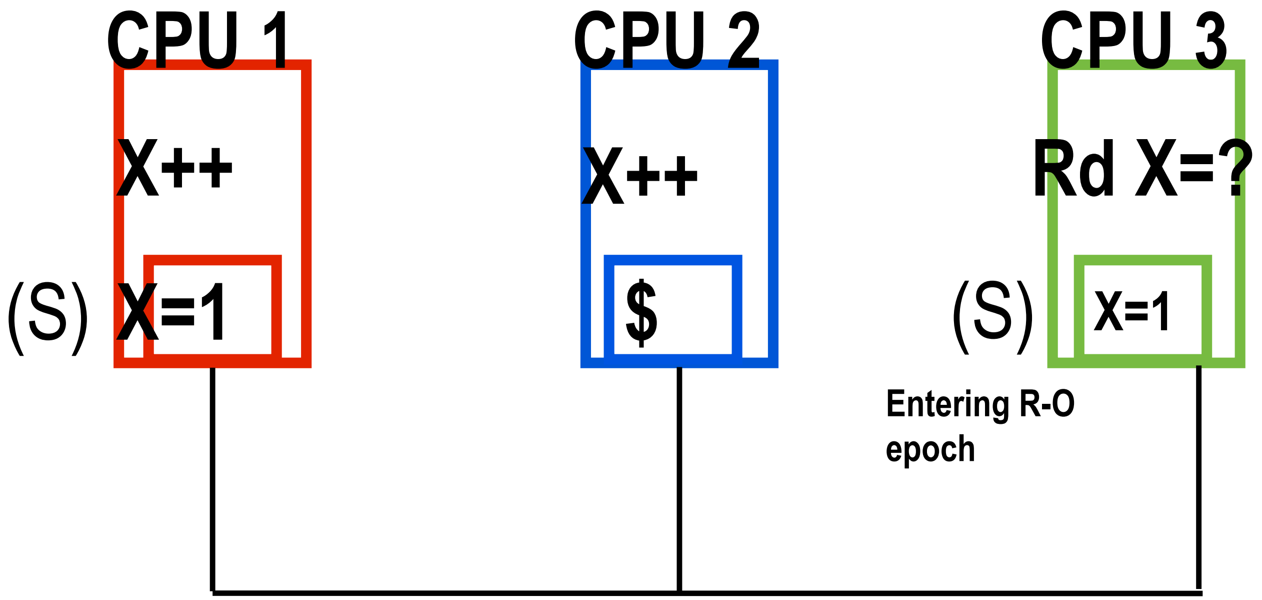
X++

Rd X=?

Implementing the Protocol



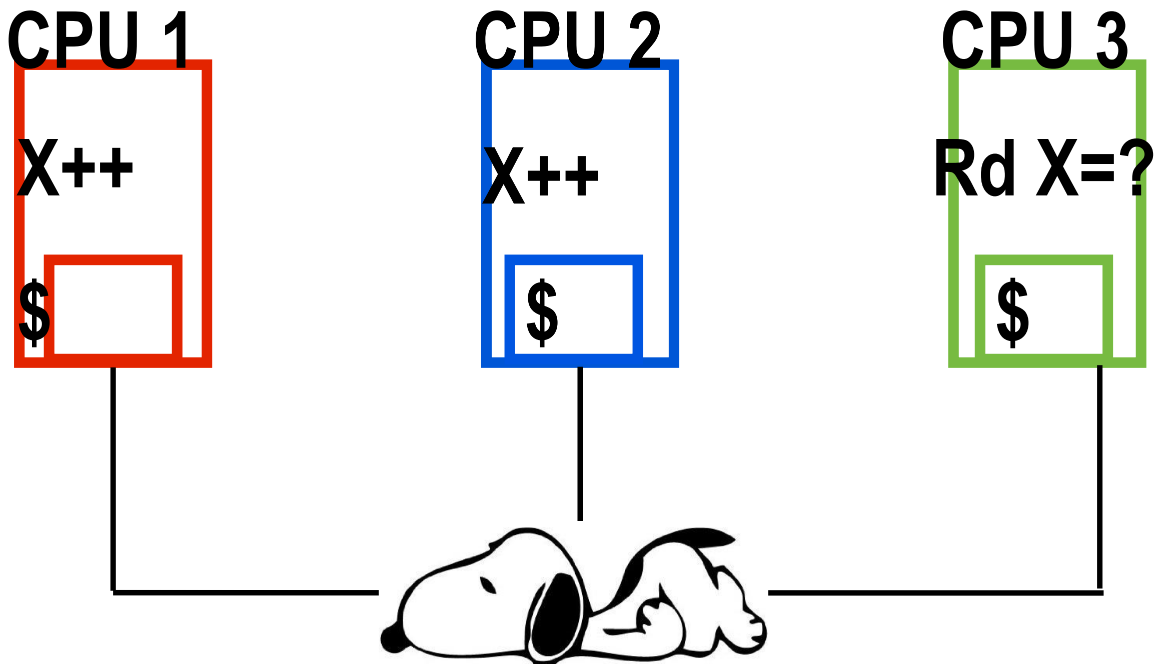
Implementing the Protocol



X++

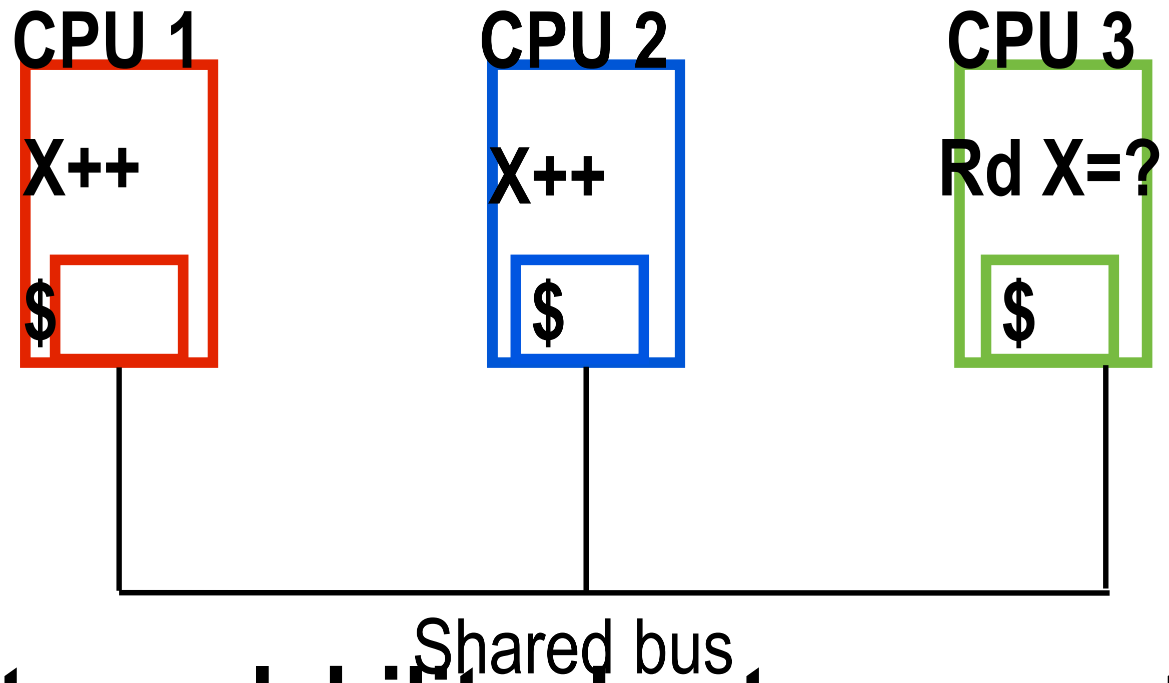
Rd X=?

Implementing the Protocol



What sucks about Snoopy?

Implementing the Protocol



Bus limits scalability due to congestion and complex message arbitration

Modern systems use a *distributed directory* to avoid this congestion

No shared medium, distributed points of arbitration for different data

Memory Consistency

Memory Consistency Model

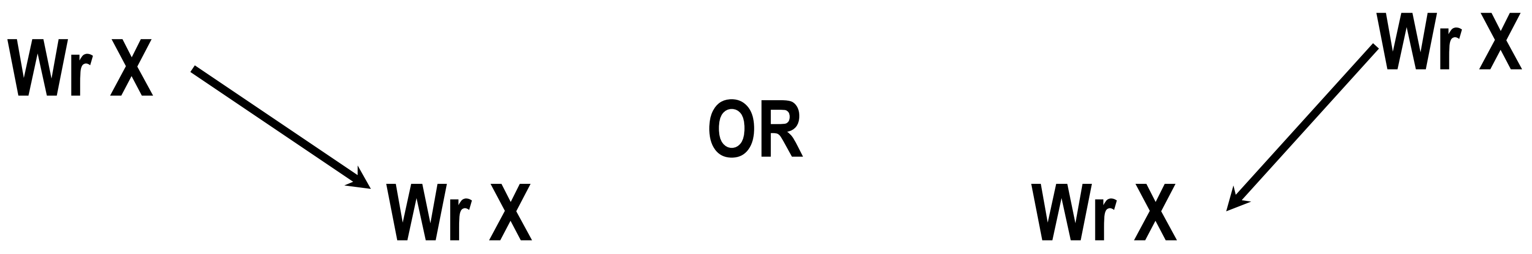
Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”

“Defines which reorderings of memory operations are permitted”

Coherence is Ordering



Coherence defines the set of legal orders of accesses to a single memory location

Consistency is Ordering

$$\begin{tikzcd} \text{Wr X} \arrow[dr] & & & & \text{Wr Y} \arrow[dl] \\ & \text{Wr Y} & \text{OR} & \text{Wr X} & \end{tikzcd}$$

Consistency defines the set of legal orders of accesses to multiple memory locations

Sequential Consistency (SC)

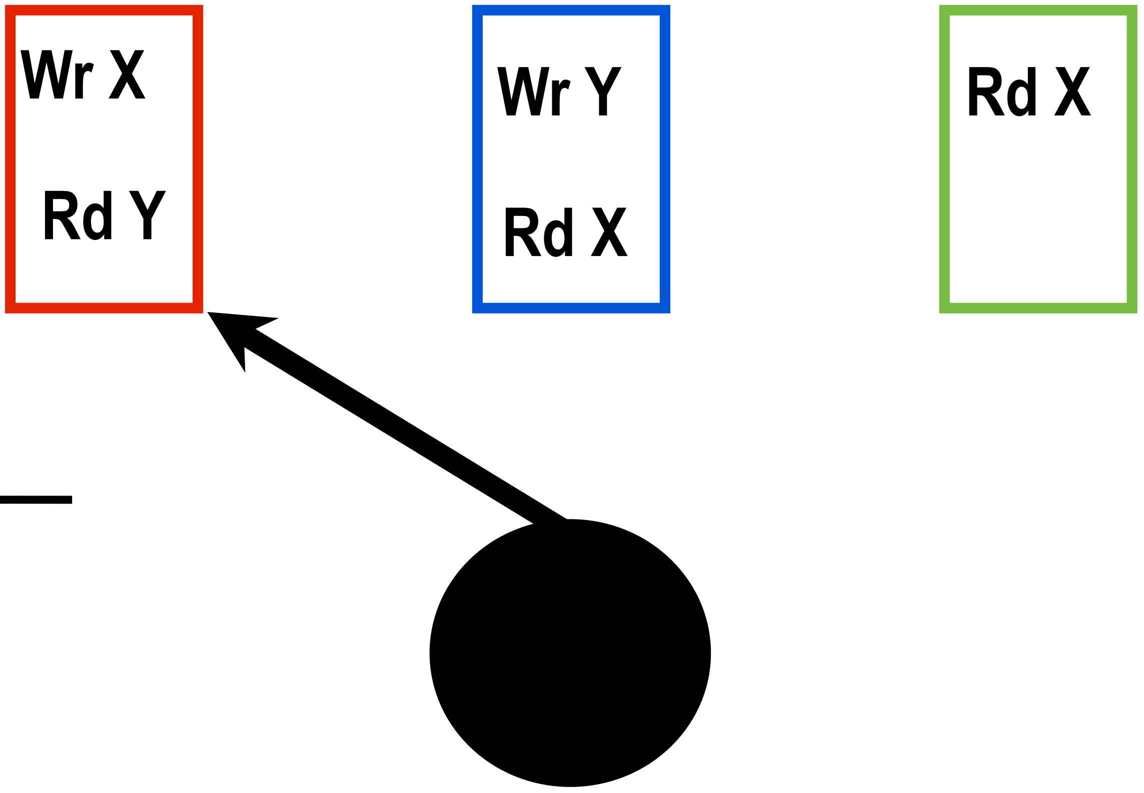
The simplest, most intuitive memory consistency model

Two Invariants to SC:

Instructions are executed in program order

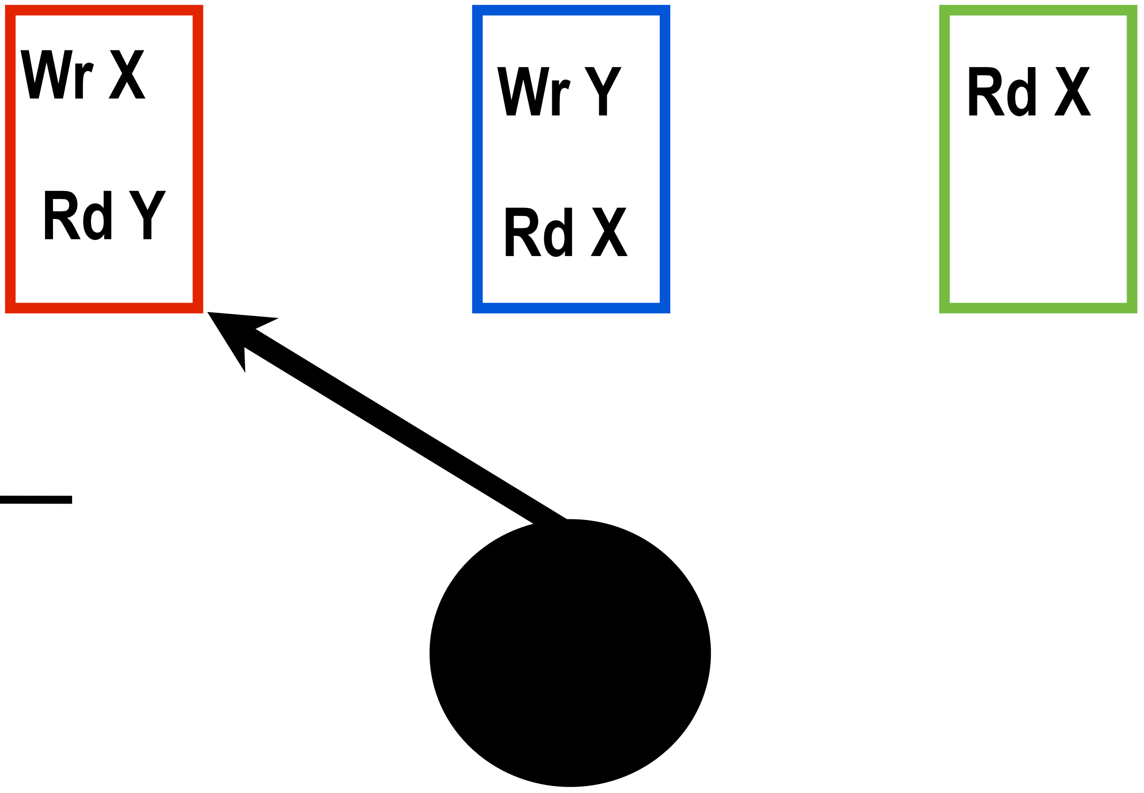
All processors agree on a total order of executed instructions

The SC "Switch"



Execution

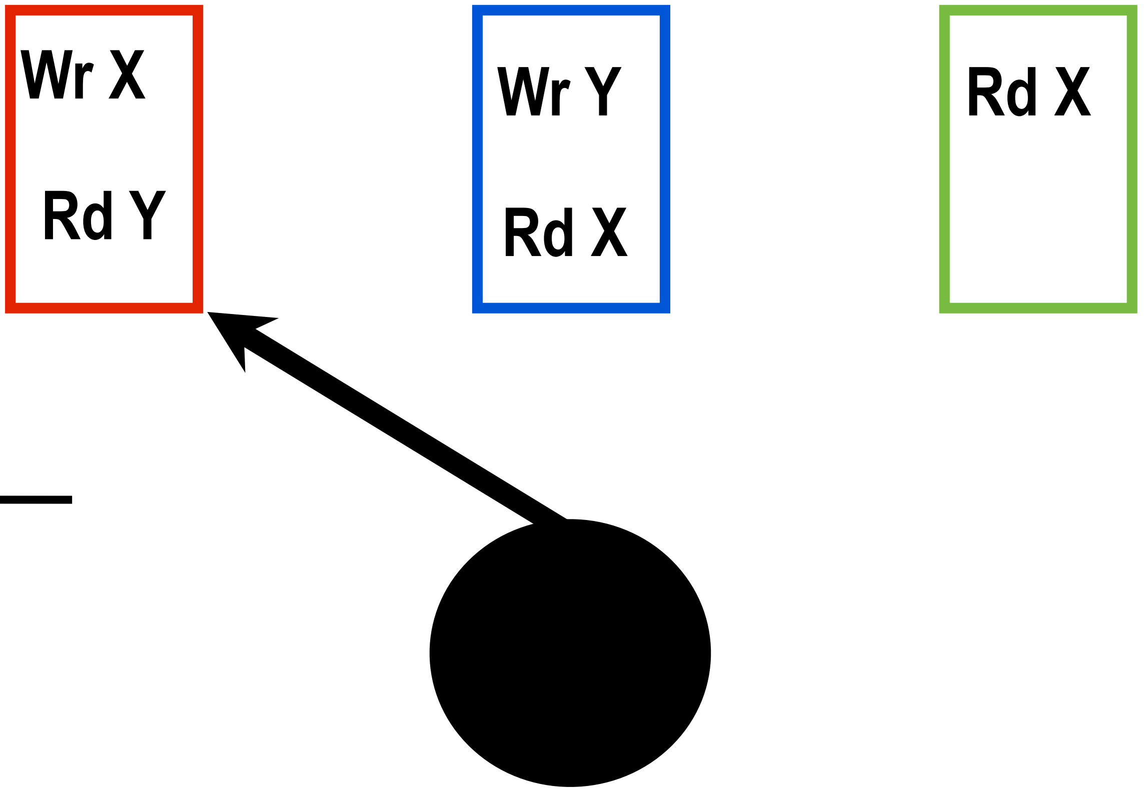
The SC "Switch"



Execution

Wr X

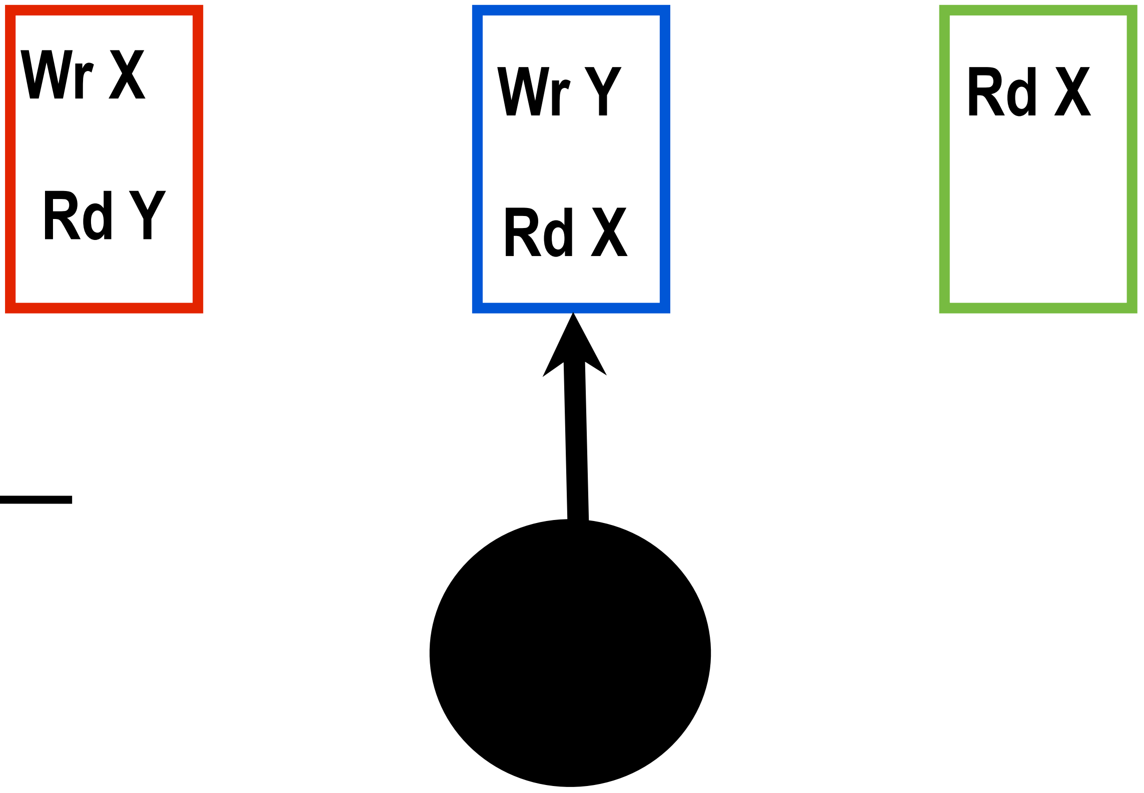
The SC "Switch"



Execution

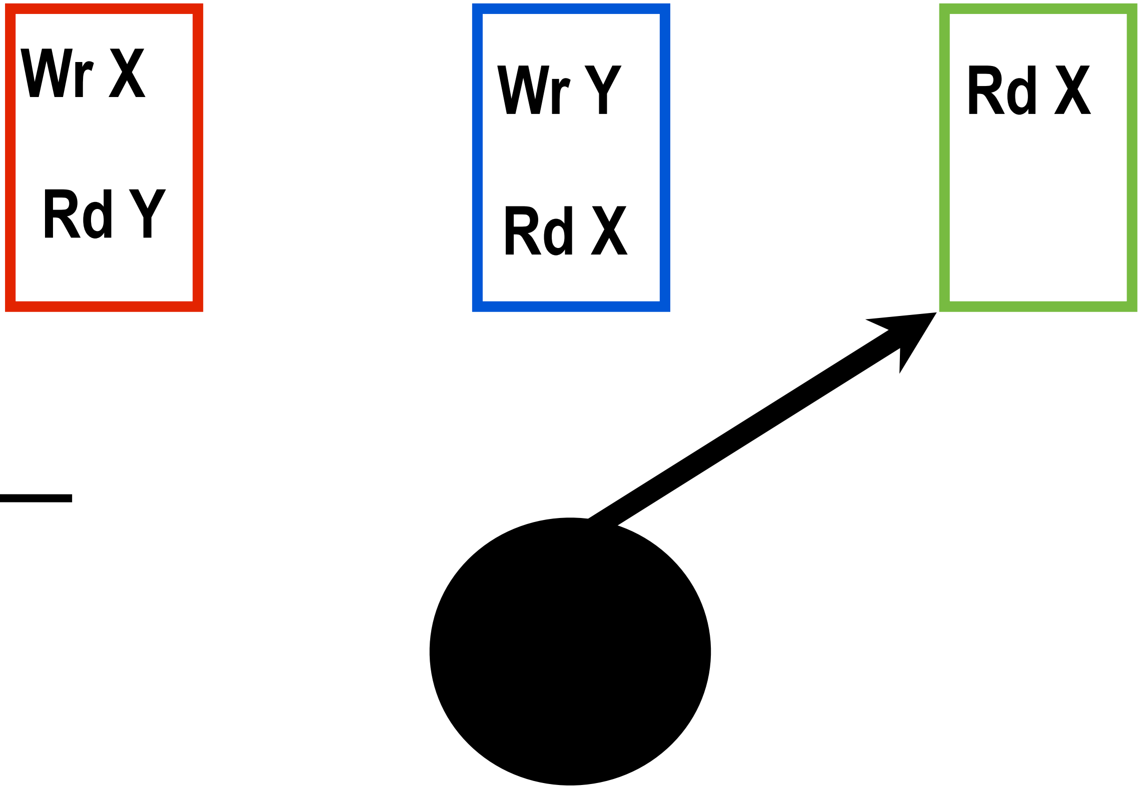
Wr X
Rd Y

The SC "Switch"



Execution
Wr X
Rd Y
Wr Y

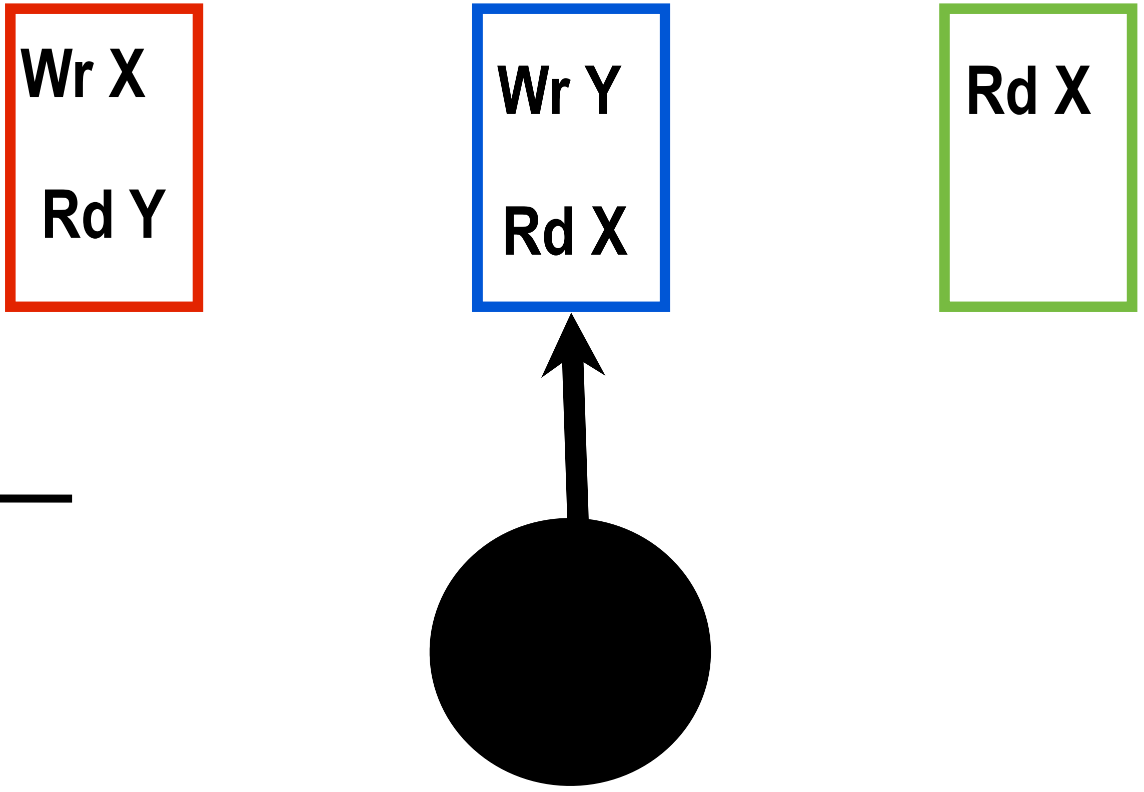
The SC “Switch”



Execution

Wr X
Rd Y
Wr Y
Rd X

The SC "Switch"



Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X

Why is SC Important?

SC is the most complex model that we can ask programmers to think about.

Wr X	Wr Y	Rd X
Rd Y	Rd X	

Intuitive (SC)	Weird (not SC)
Wr X	Rd Y
Rd Y	Wr X
Wr Y	Rd X
Rd X	Rd X
Rd X	Wr Y

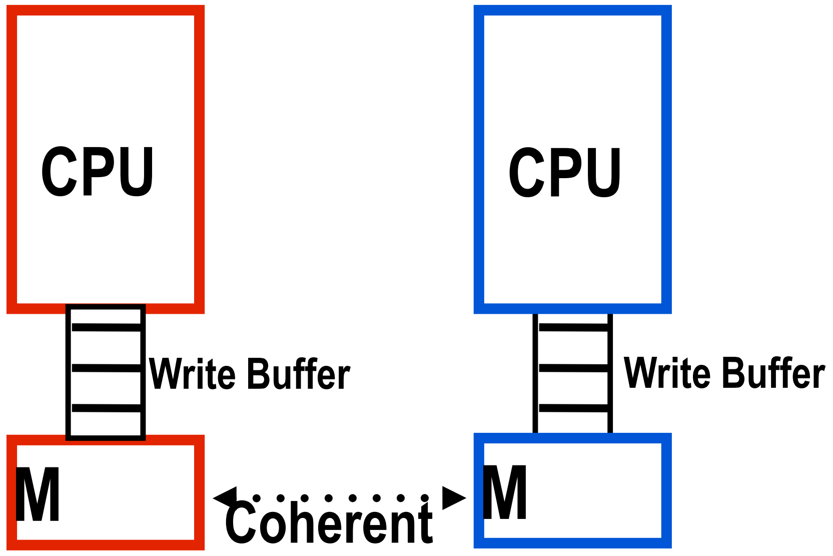
SC prohibits all reordering of instructions (Invariant 1)

Why are Instructions Reordered?

Why are Instructions Reordered?

Optimization.

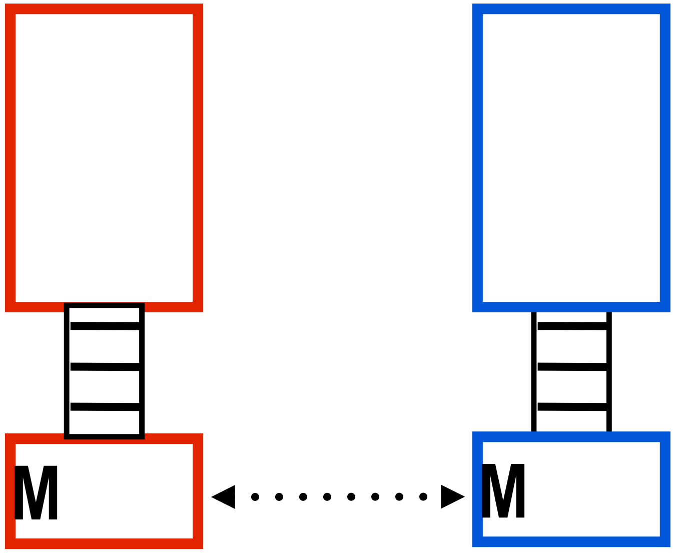
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



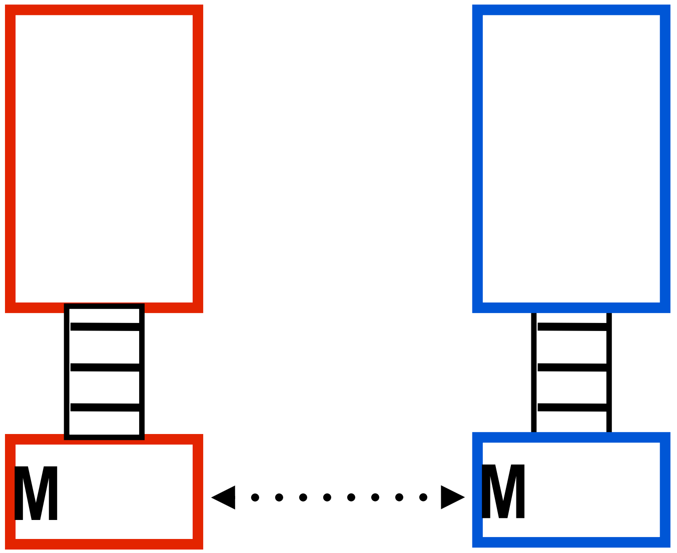
Program

Initially X == Y == 0

X=1	Y=1
r1=Y	r2=X

Is r1==r2==0
a valid result?

Reordering #1: Write Buffers



Program

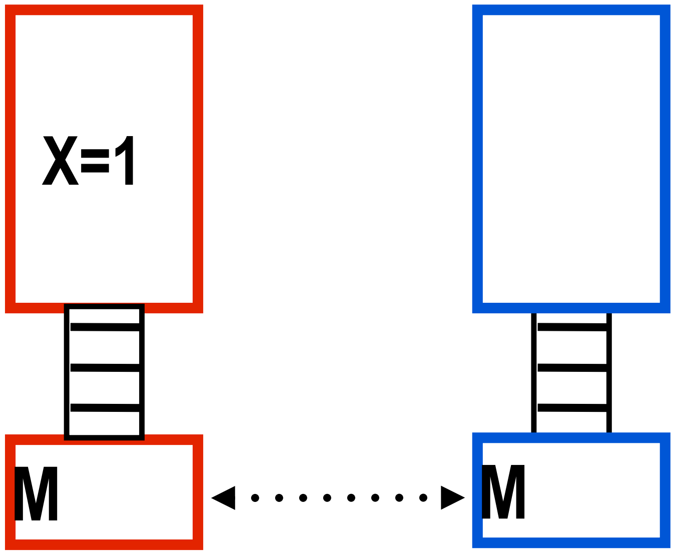
Initially X == Y == 0

X=1	Y=1
r1=Y	r2=X

Is r1==r2==0 a valid result?

r1 == r2 == 0 is not SC, but it can happen with write buffers

Reordering #1: Write Buffers



Program

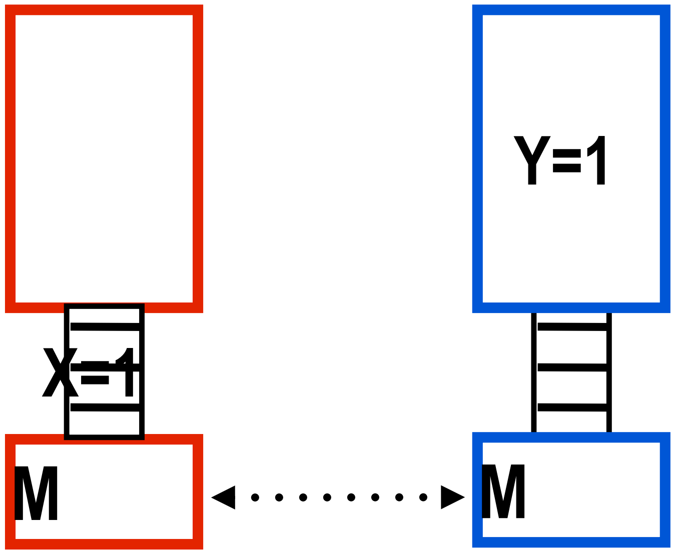
Initially X == Y == 0

Y=1

r1=Y r2=X

Execution

Reordering #1: Write Buffers



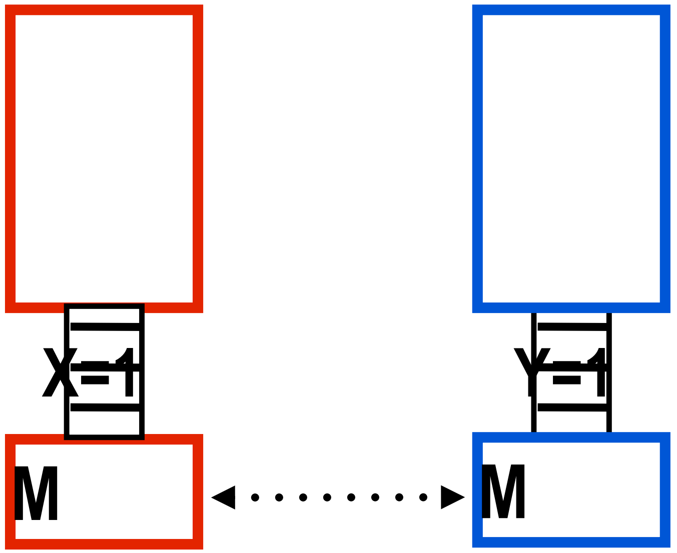
Program

Initially X == Y == 0

r1=Y   r2=X

Execution

Reordering #1: Write Buffers



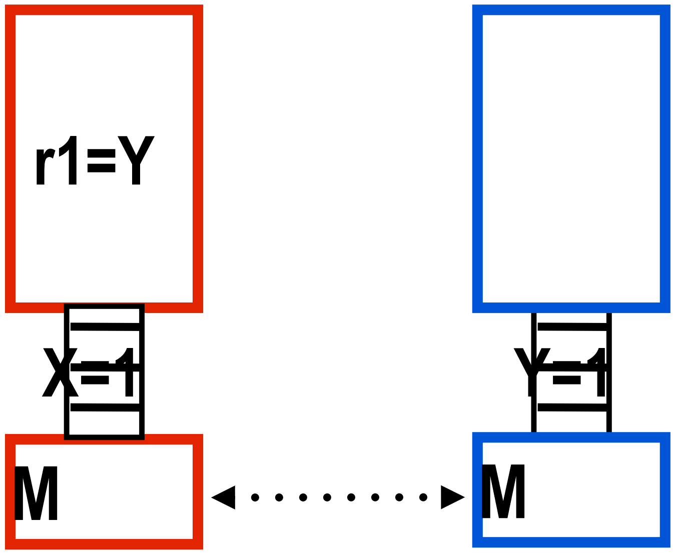
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



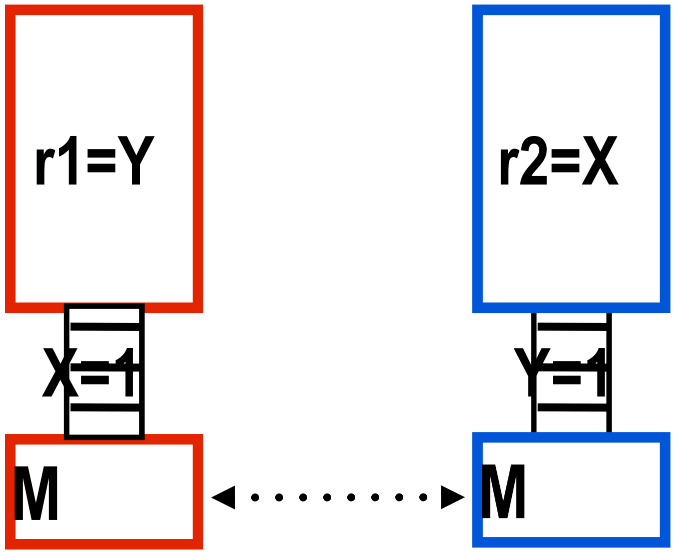
Program

Initially X == Y == 0

r2=X

Execution

Reordering #1: Write Buffers

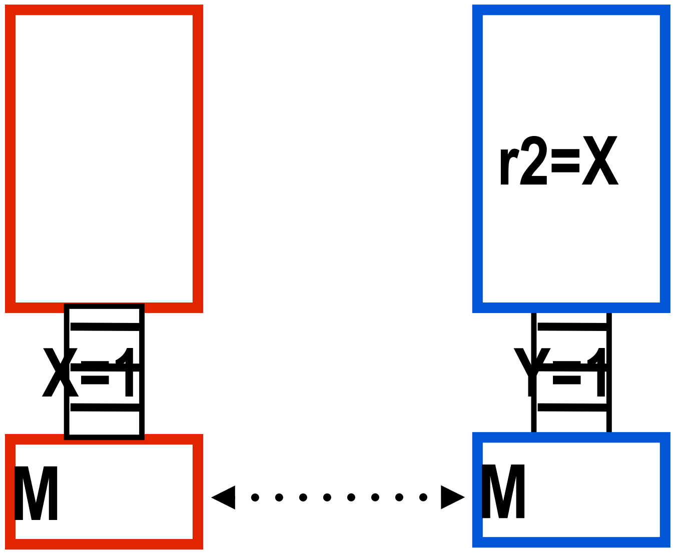


Program

Initially X == Y == 0

Execution

Reordering #1: Write Buffers



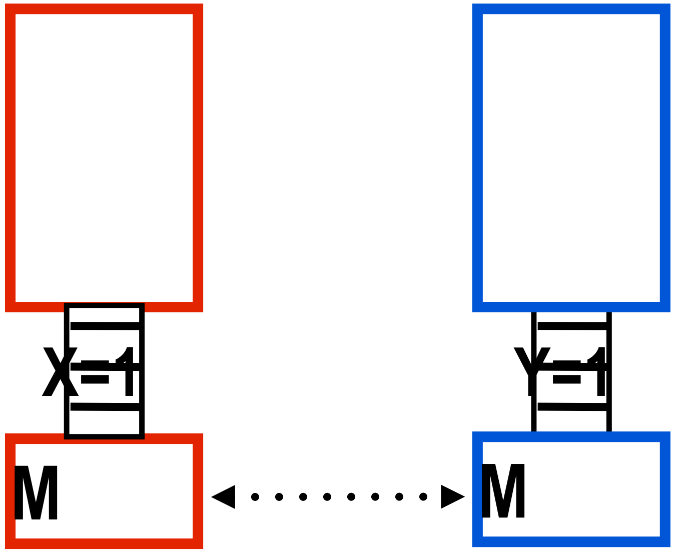
Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

Reordering #1: Write Buffers



Program

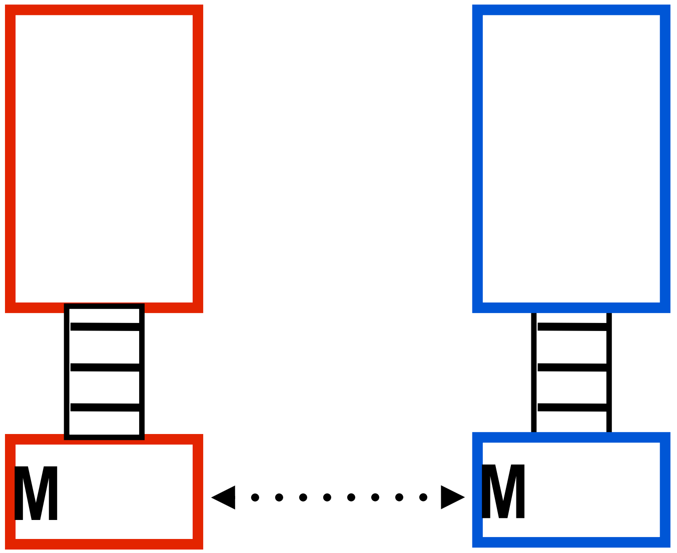
Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

Reordering #1: Write Buffers



WBs let reads finish before older writes

Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]
r2=X [r2 <- 0]
X=1
Y=1 (Not SC!)

When is an Execution Not SC?

When a memory operation happens before itself

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

X=1

Y=1

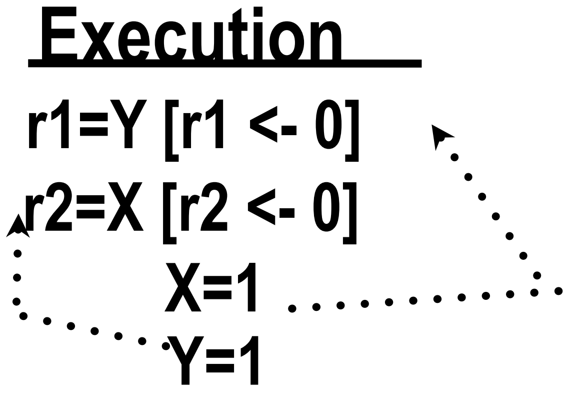
Happens-Before Graph

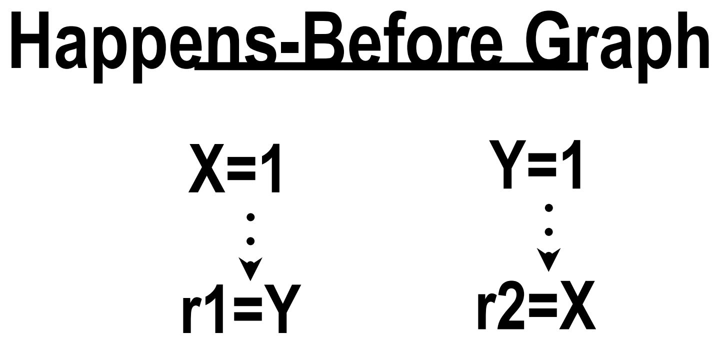
X=1 Y=1

r1=Y r2=X

When is an Execution Not SC?

When a memory operation happens before itself



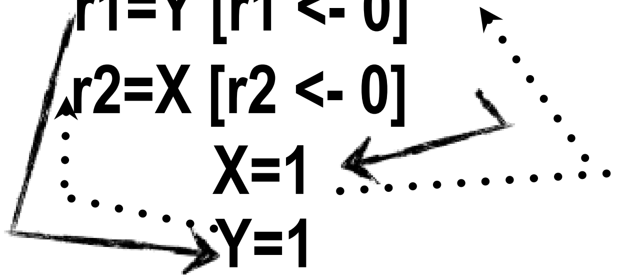


Program Order HB Edge

When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph

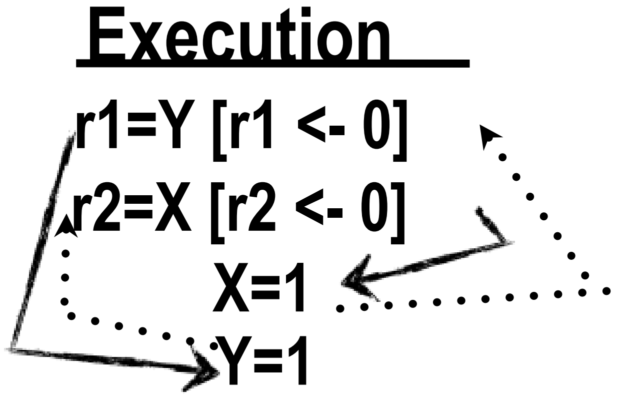
$$\begin{tikzcd}
X=1 \arrow[d, dotted] & Y=1 \arrow[d, dotted] \\
r1=Y \arrow[ur] & r2=X \arrow[ul]
\end{tikzcd}$$

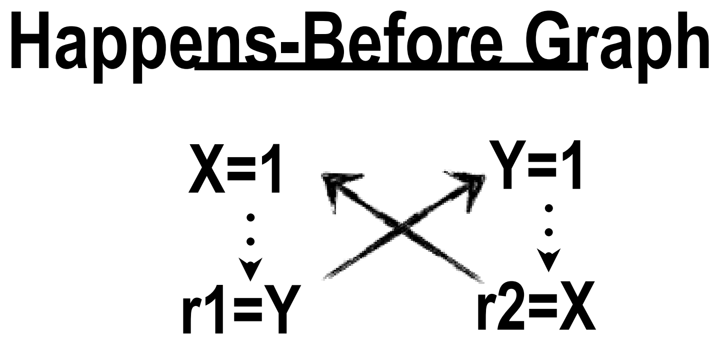
Program Order HB Edge

Causal Order HB Edge

When is an Execution Not SC?

When a memory operation happens before itself





If there is a cycle in the happens-before graph, the execution is not SC