

# **CSC 252: Computer Organization**

## **Spring 2018: Lecture 26**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

### **Action Items:**

- **Programming Assignment 4 grades out**
- **Programming Assignment 5 re-grade open**
- **Programming Assignment 6 due soon**

# Announcement

- Programming assignment 6 is due on 11:59pm, **Monday, April 30**.
- Programming assignment 5 re-grade is open until 11:59pm, Friday
- Programming assignment 4 grades are out

22	23	24	25	26	27	28
29	30	May 1	2	3	4	5

**Due**

**Last  
Lecture**

# Today

- Shared variables in multi-threaded programming
  - Mutual exclusion using semaphore
  - Deadlock
- Thread-level parallelism
  - Amdahl's Law: performance model of parallel programs
- Hardware support for multi-threading
  - Single-core
  - Hyper-threading
  - Multi-core
  - Cache coherence

# Binary Semaphore Protecting Critical Section

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;           /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with P and V:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

goodcnt.c

# Deadlock

- Def: A process/thread is *deadlocked* if and only if it is waiting for a condition that will never be true
- General to concurrent/parallel programming (threads, processes)
- Typical Scenario
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!

# Deadlocking With Semaphores

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}

int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

**Tid[0]:**  
P(s<sub>0</sub>);  
P(s<sub>1</sub>);  
cnt++;  
V(s<sub>0</sub>);  
V(s<sub>1</sub>);

**Tid[1]:**  
P(s<sub>1</sub>);  
P(s<sub>0</sub>);  
cnt++;  
V(s<sub>1</sub>);  
V(s<sub>0</sub>);

# Avoiding Deadlock

*Acquire shared resources in same order*

```
Tid[0]:  
P(s0);  
P(s1);  
cnt++;  
V(s0);  
V(s1);
```

```
Tid[1]:  
P(s1);  
P(s0);  
cnt++;  
V(s1);  
V(s0);
```



```
Tid[0]:  
P(s0);  
P(s1);  
cnt++;  
V(s0);  
V(s1);
```

```
Tid[1]:  
P(s0);  
P(s1);  
cnt++;  
V(s1);  
V(s0);
```

# Another Deadlock Example: Signal Handling

- Signal handlers are concurrent with main program and may share the same global data structures.



# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- OS decides to take the SIGCHLD interrupt and executes the handler

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- OS decides to take the SIGCHLD interrupt and executes the handler
- When return to parent process, **y == 20!**

# Fixing the Signal Handling Bug

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);

    exit(0);
}
```

- Block all signals before accessing a shared, global data structure.

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.



# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.
- Signal handler wants the mutex, which is acquired by the main program.

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.
- Signal handler wants the mutex, which is acquired by the main program.
- **Key:** signal handler is in the same process as the main program. The kernel forces the handler to finish before returning to the main program.

# Summary of Multi-threading Programming

- Concurrent/parallel threads access shared variables
- Need to protect concurrent accesses to guarantee correctness
- Semaphores (e.g., mutex) provide a simple solution
- Can lead to deadlock if not careful
- Take CSC 258 to know more about avoiding deadlocks (and parallel programming in general)

# Thinking in Parallel is Hard

**Thinking in Parallel is Hard**

**Maybe Thinking is Hard**

# Today

- Shared variables in multi-threaded programming
  - Mutual exclusion using semaphore
  - Deadlock
- **Thread-level parallelism**
  - Amdahl's Law: performance model of parallel programs
- Hardware support for multi-threading
  - Single-core
  - Hyper-threading
  - Multi-core
  - Cache coherence

# Thread-level Parallelism (TLP)

- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task
- Example: Parallel summation of  $N$  number
  - Should add up to  $((n-1)*n)/2$
- Partition values  $1, \dots, n-1$  into  $t$  ranges
  - $\lfloor n/t \rfloor$  values in each range
  - Each of  $t$  threads processes one range (sub-task)
  - Sum all sub-sums in the end

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up



# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$1 - f$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$1 - f +$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$1 - f + \frac{f}{N}$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable ( $f = 1$ ): Speedup =  $N$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable ( $f = 1$ ): Speedup =  $N$
- Completely sequential ( $f = 0$ ): Speedup = 1

# Amdahl's Law

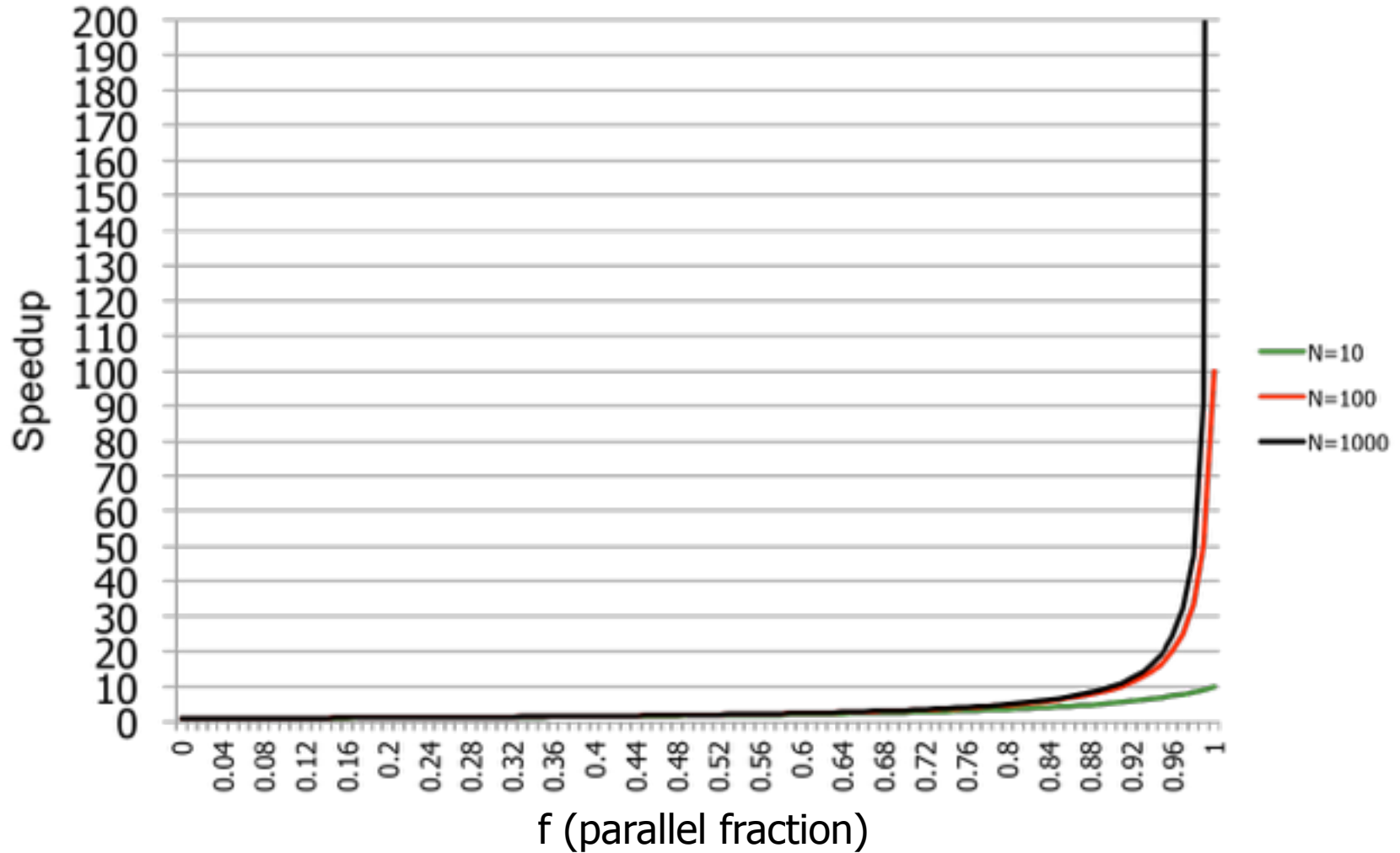
- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable ( $f = 1$ ): Speedup =  $N$
- Completely sequential ( $f = 0$ ): Speedup = 1
- Mostly parallelizable ( $f = 0.9$ ,  **$N = 1000$** ): **Speedup = 9.9**



# Sequential Bottleneck



# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {  
    Compute  
    P(A)  
    Update shared data  
    V(A)  
}
```

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {  
  Compute            N  
  P(A)  
  Update shared data  
  V(A)  
}
```

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {  
    Compute N  
    P(A)  
    Update shared data  
    V(A) C  
}
```

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {
```

```
  Compute
```

**N**

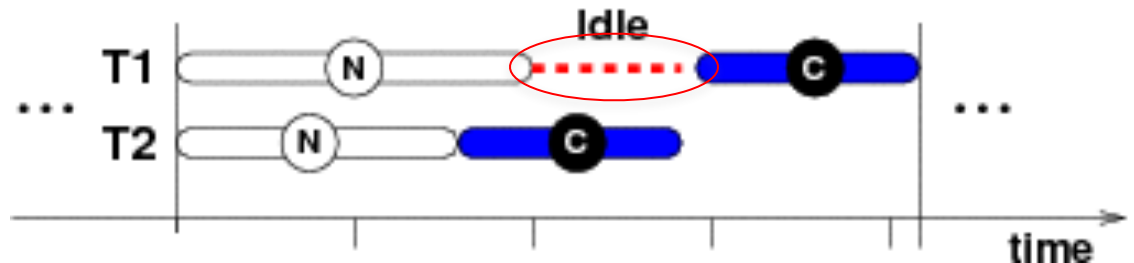
```
  P(A)
```

```
    Update shared data
```

```
  V(A)
```

**C**

```
}
```



# Today

- Shared variables in multi-threaded programming
  - Mutual exclusion using semaphore
  - Deadlock
- Thread-level parallelism
  - Amdahl's Law: performance model of parallel programs
- **Hardware support for multi-threading**
  - Single-core
  - Hyper-threading
  - Multi-core
  - Cache coherence



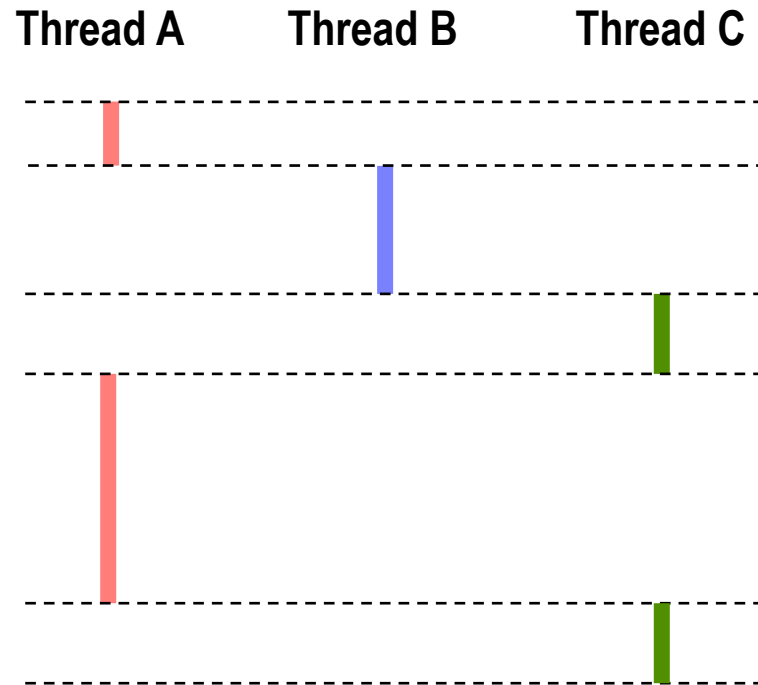
# Can A Single Core Support Multi-threading?

- Need to multiplex between different threads (time slicing)

## Sequential

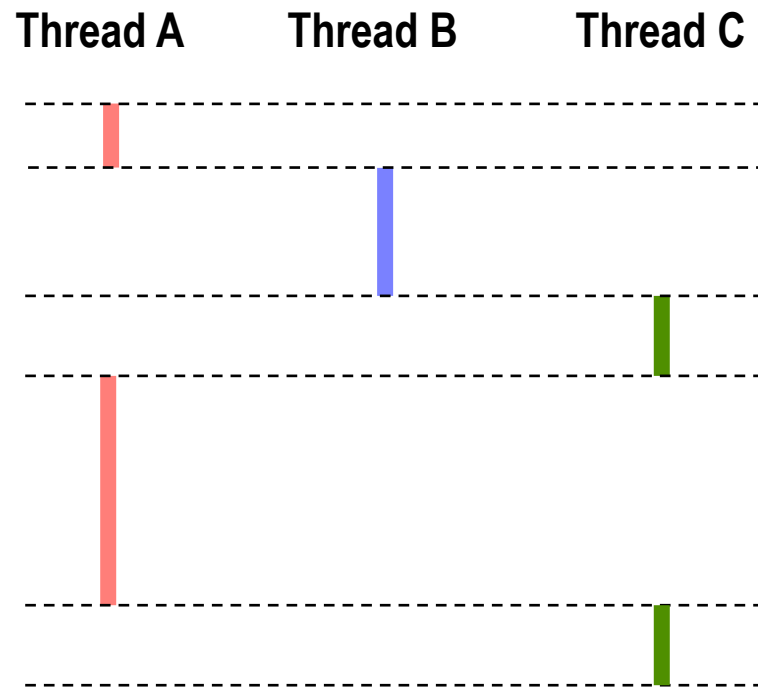


## Multi-threaded



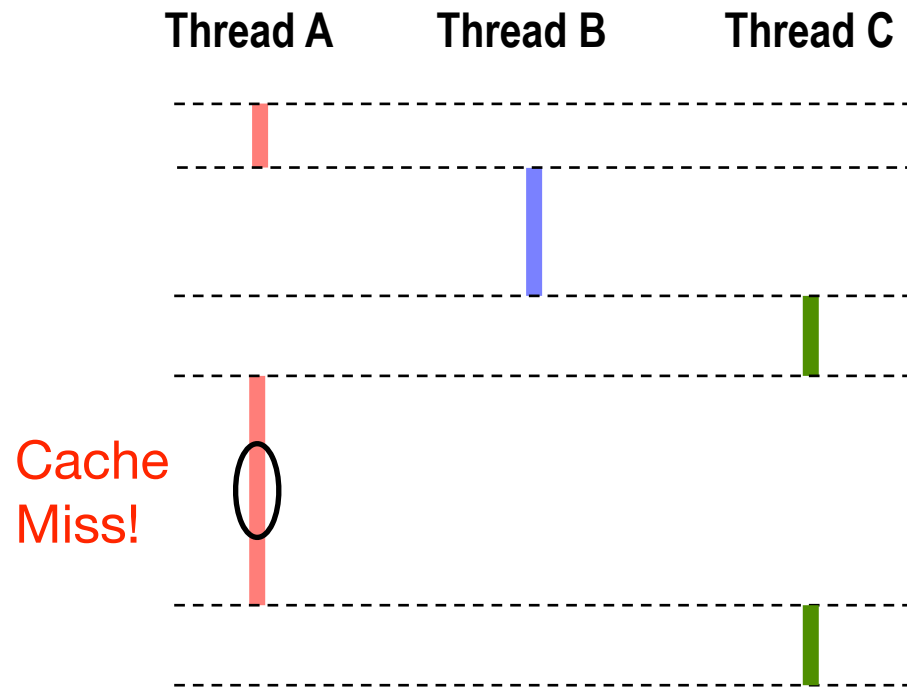
# Any benefits?

- Can single-core multi-threading provide any performance gains?



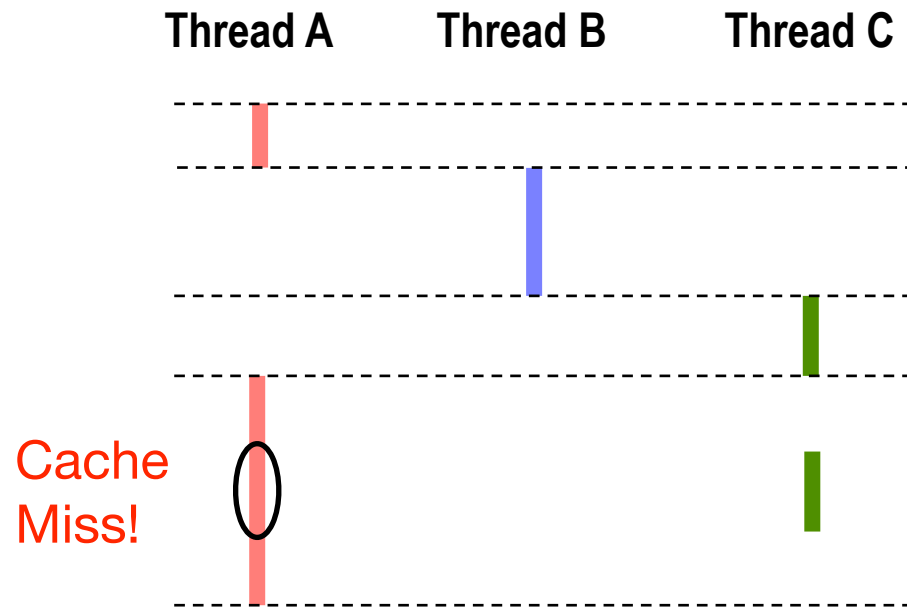
# Any benefits?

- Can single-core multi-threading provide any performance gains?



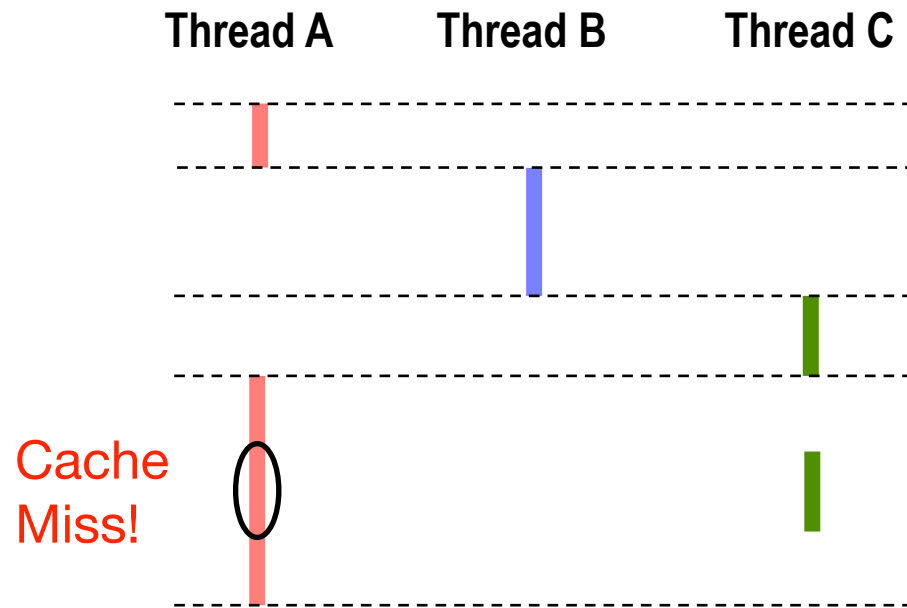
# Any benefits?

- Can single-core multi-threading provide any performance gains?



# Any benefits?

- Can single-core multi-threading provide any performance gains?
- If Thread A has a cache miss and the pipeline gets stalled, switch to Thread C. Improves the overall performance.



# When to Switch?

- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)

# When to Switch?

- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)
- Fine grained
  - Cycle by cycle
  - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
  - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. Seminal paper that shows that using multi-threading can avoid branch prediction.

# When to Switch?

- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)
- Fine grained
  - Cycle by cycle
  - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
  - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. Seminal paper that shows that using multi-threading can avoid branch prediction.
- Either way, need to save/restore thread context upon switching

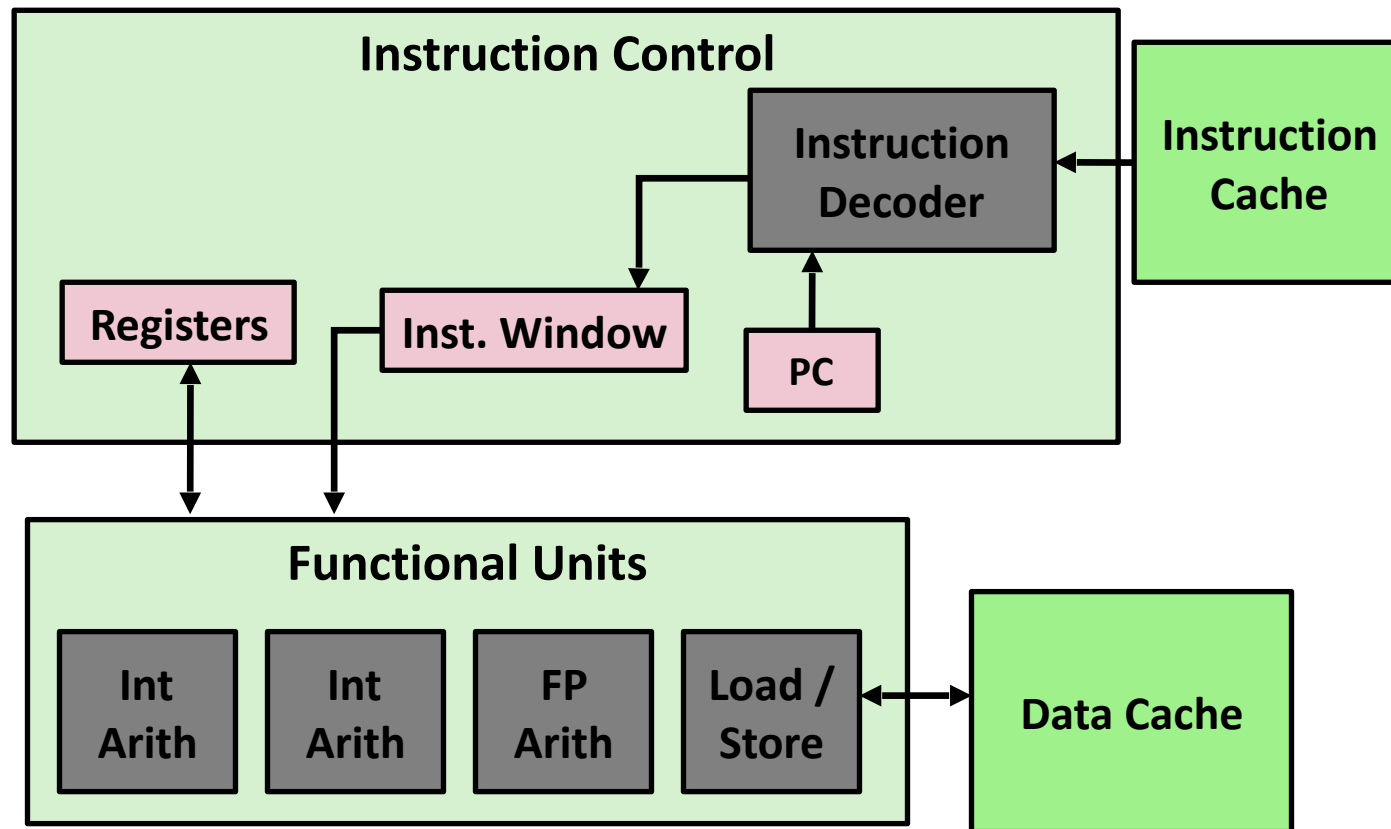


# Today

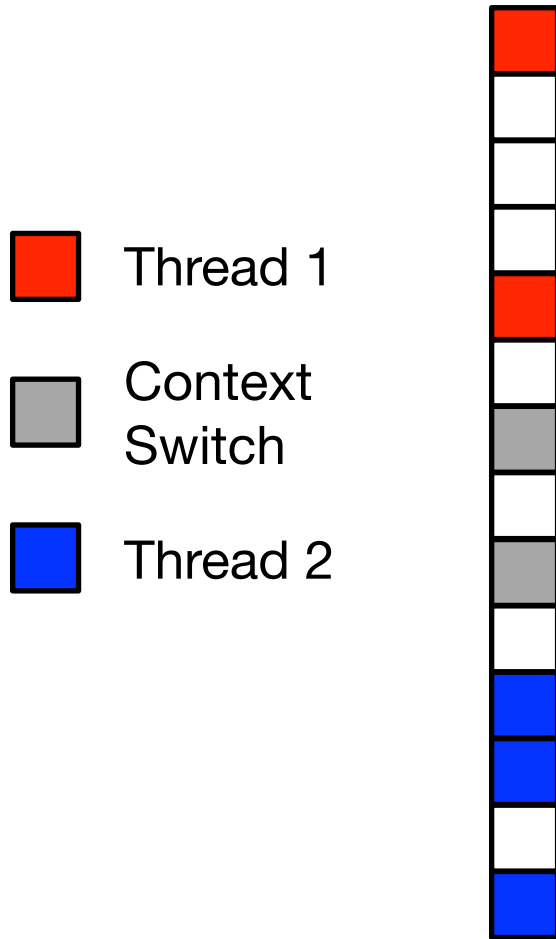
- Shared variables in multi-threaded programming
  - Mutual exclusion using semaphore
  - Deadlock
- Thread-level parallelism
  - Amdahl's Law: performance model of parallel programs
- **Hardware support for multi-threading**
  - Single-core
  - Hyper-threading
  - Multi-core
  - Cache coherence

# Single-Core Internals

- Typically has multiple function units to allow for issuing multiple instructions at the same time
- Called “Superscalar” Microarchitecture

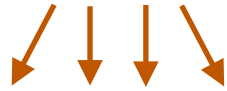





# Conventional Multi-threading

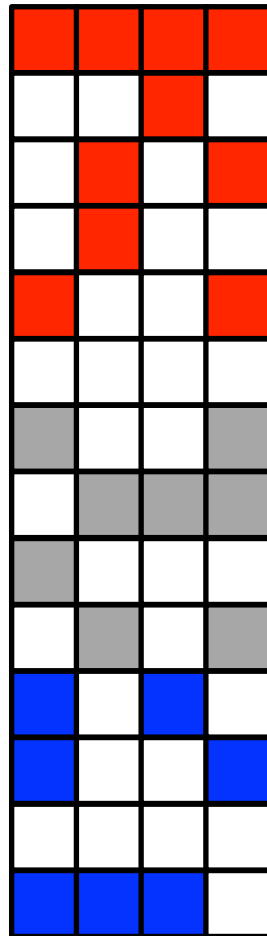


# Conventional Multi-threading

Functional Units

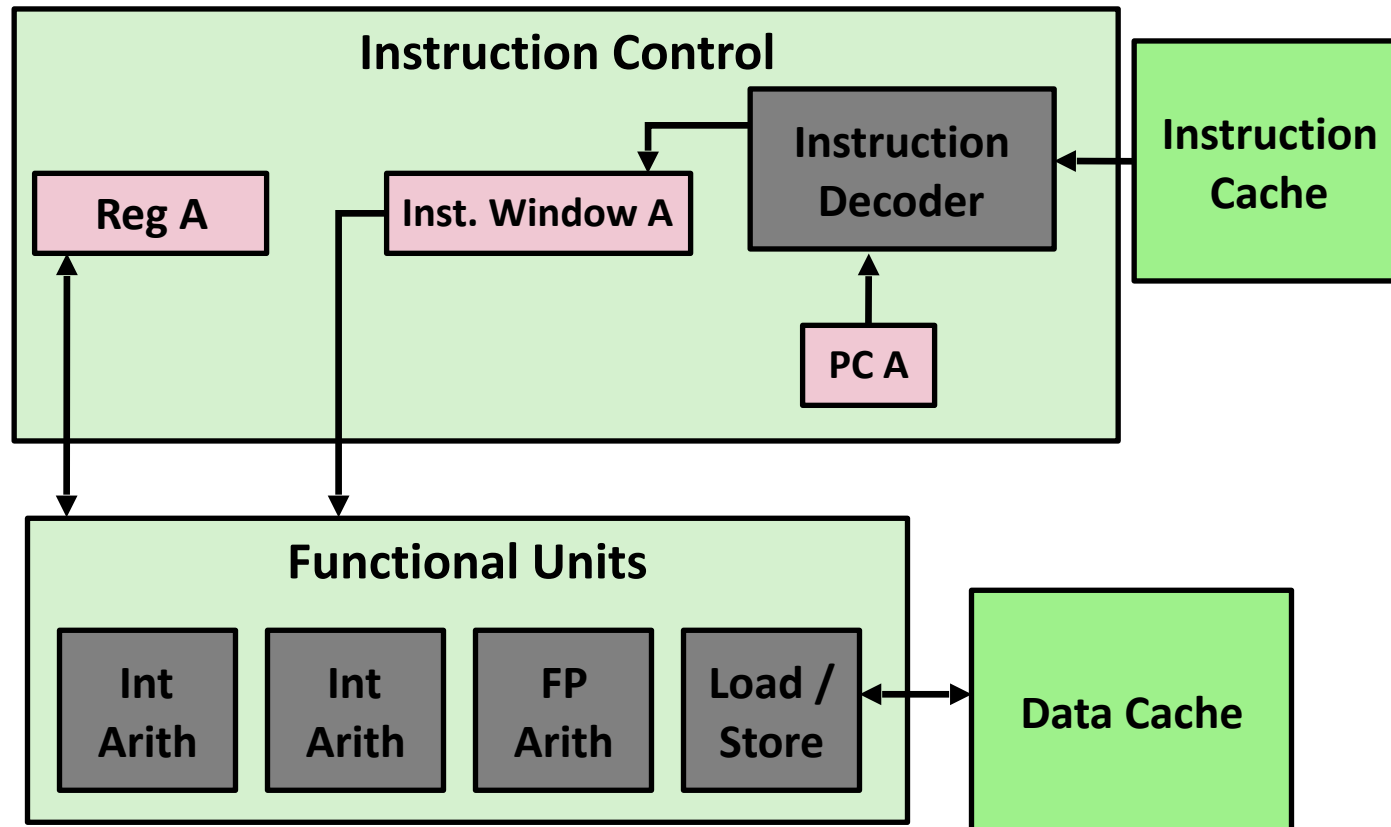


-  Thread 1
-  Context Switch
-  Thread 2



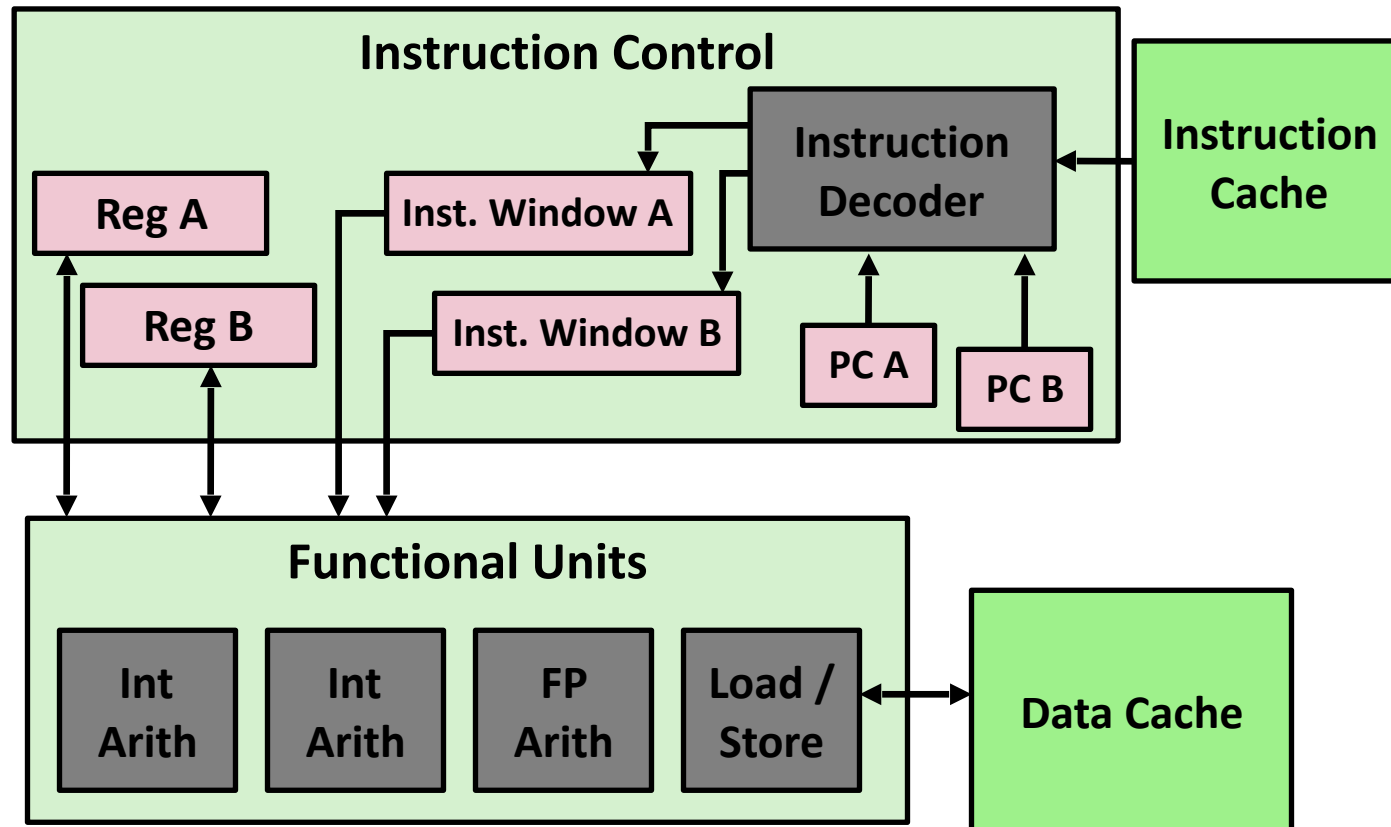
# Hyper-threading

- Intel's terminology. More commonly known as: Simultaneous Multi-threading (SMT)
- Replicate enough hardware structures to process K instruction streams
- K copies of all registers. Share functional units



# Hyper-threading

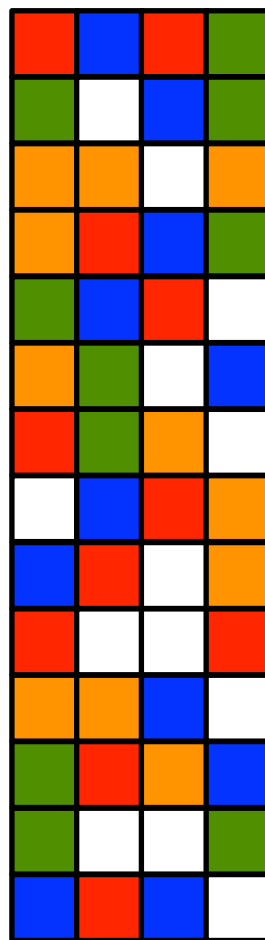
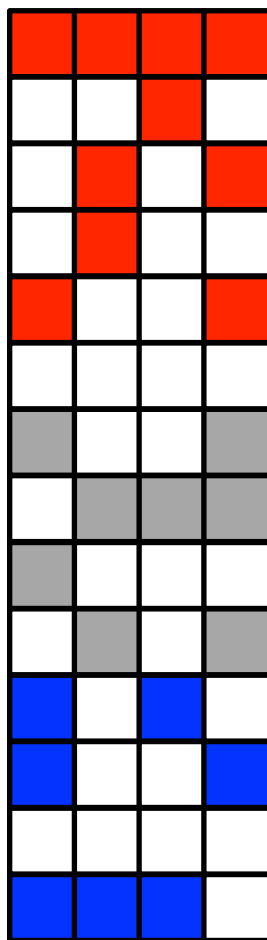
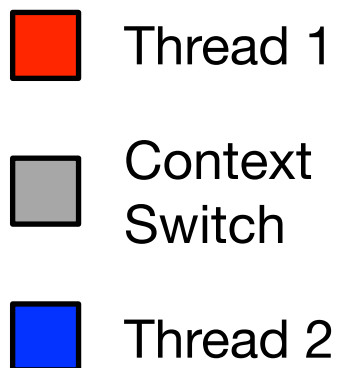
- Intel's terminology. More commonly known as: Simultaneous Multi-threading (SMT)
- Replicate enough hardware structures to process K instruction streams
- K copies of all registers. Share functional units



# Conventional Multi-threading vs. Hyper-threading

Conventional  
Multi-threading

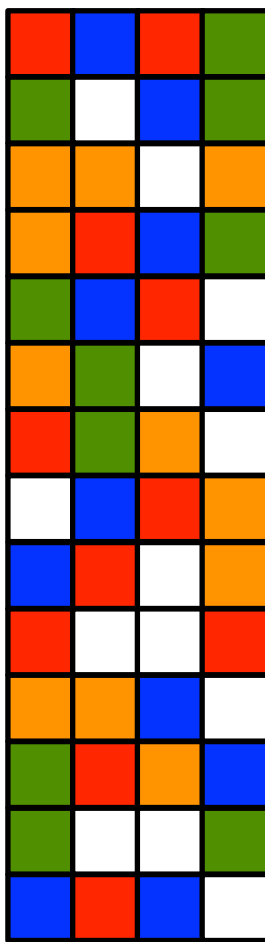
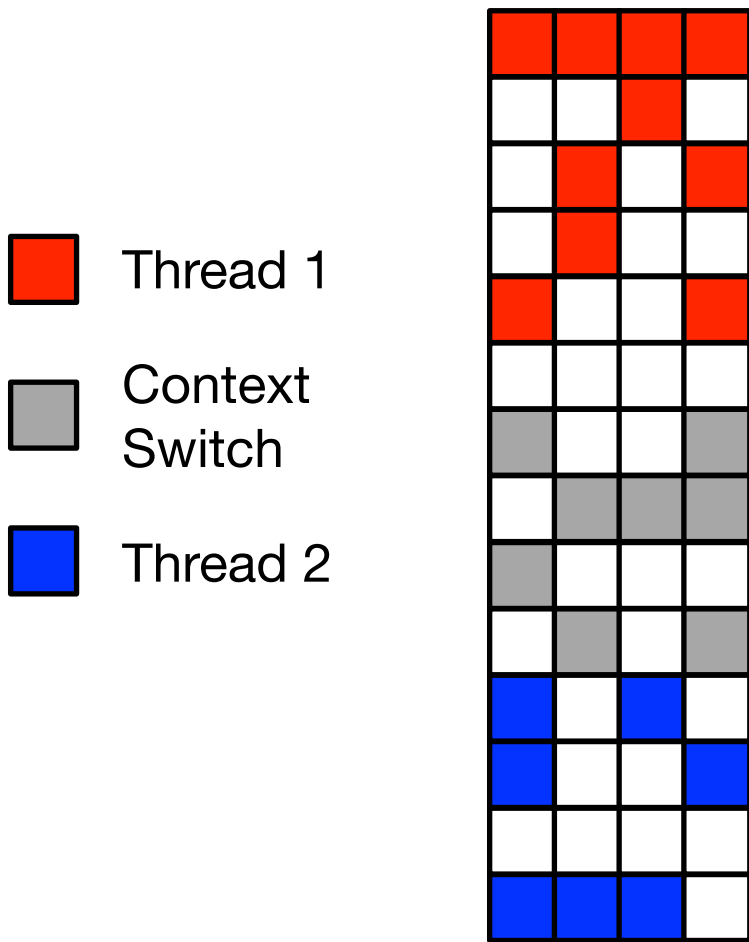
Hyper-threading



# Conventional Multi-threading vs. Hyper-threading

# Conventional Multi-threading

# Hyper-threading



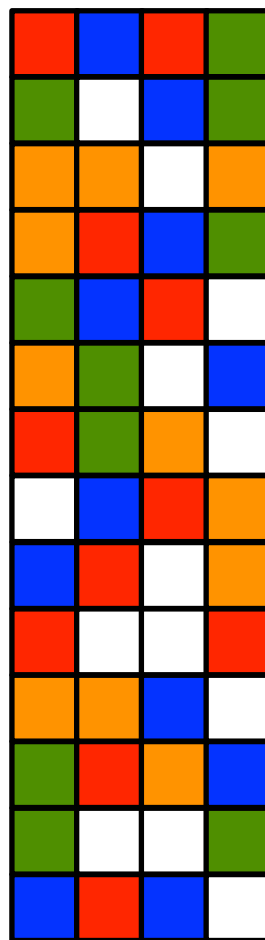
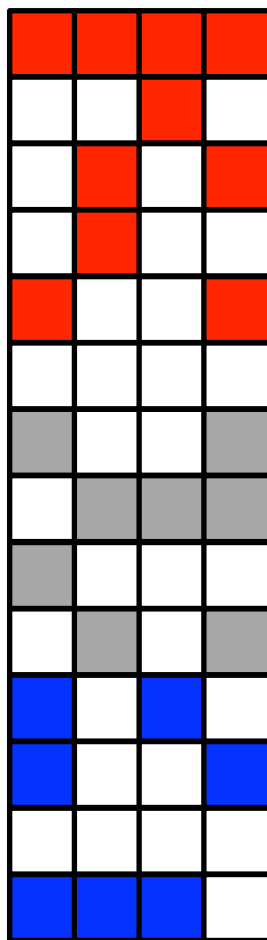
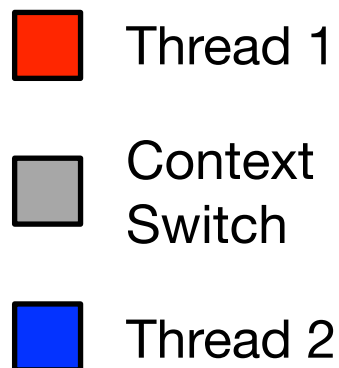
Multiple threads actually execute in parallel (even with one single core)



# Conventional Multi-threading vs. Hyper-threading

Conventional  
Multi-threading

Hyper-threading



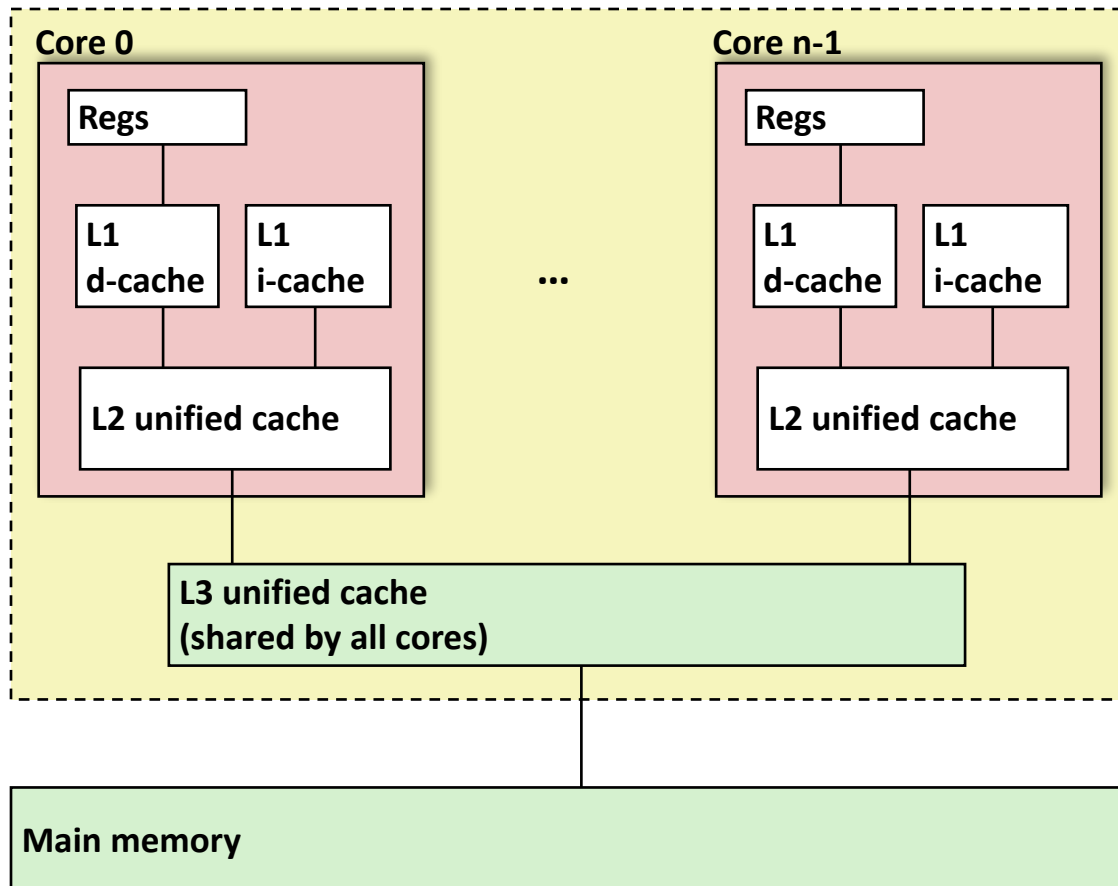
Multiple threads actually execute in parallel (even with one single core)

No/little context switch overhead

# Today

- Shared variables in multi-threaded programming
  - Mutual exclusion using semaphore
  - Deadlock
- Thread-level parallelism
  - Amdahl's Law: performance model of parallel programs
- **Hardware support for multi-threading**
  - Single-core
  - Hyper-threading
  - **Multi-core**
  - Cache coherence

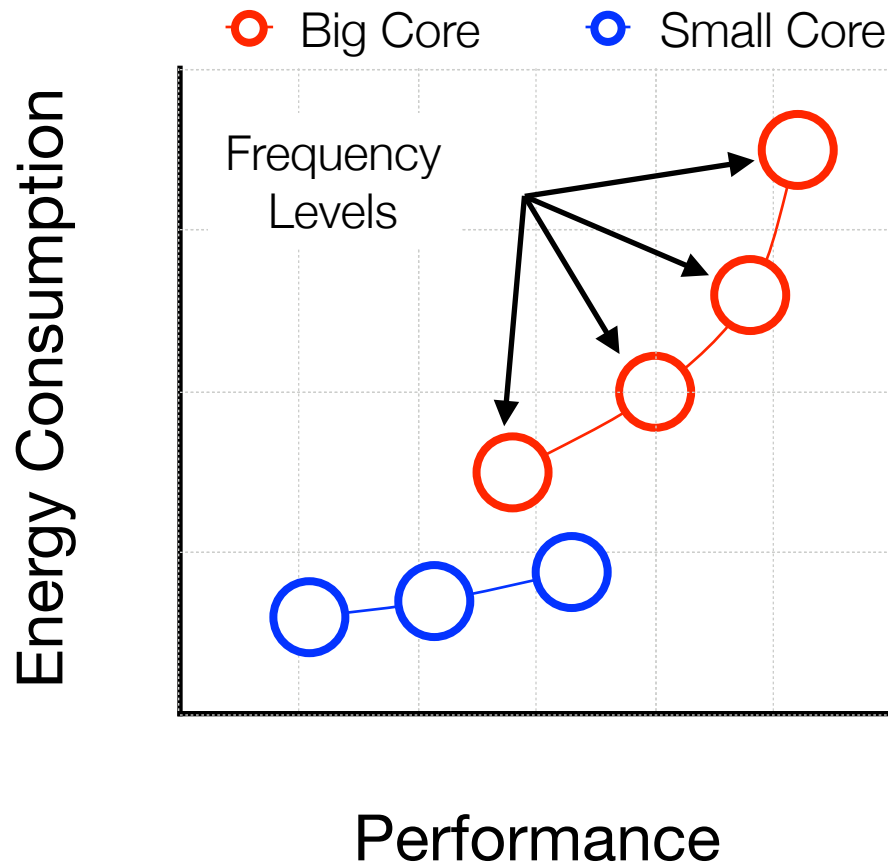
# Typical Multi-core Processor



- Traditional multiprocessing: symmetric multiprocessor (SMP)
- Every core is exactly the same. Private registers, L1/L2 caches, etc.
- Share L3 (LLC) and main memory

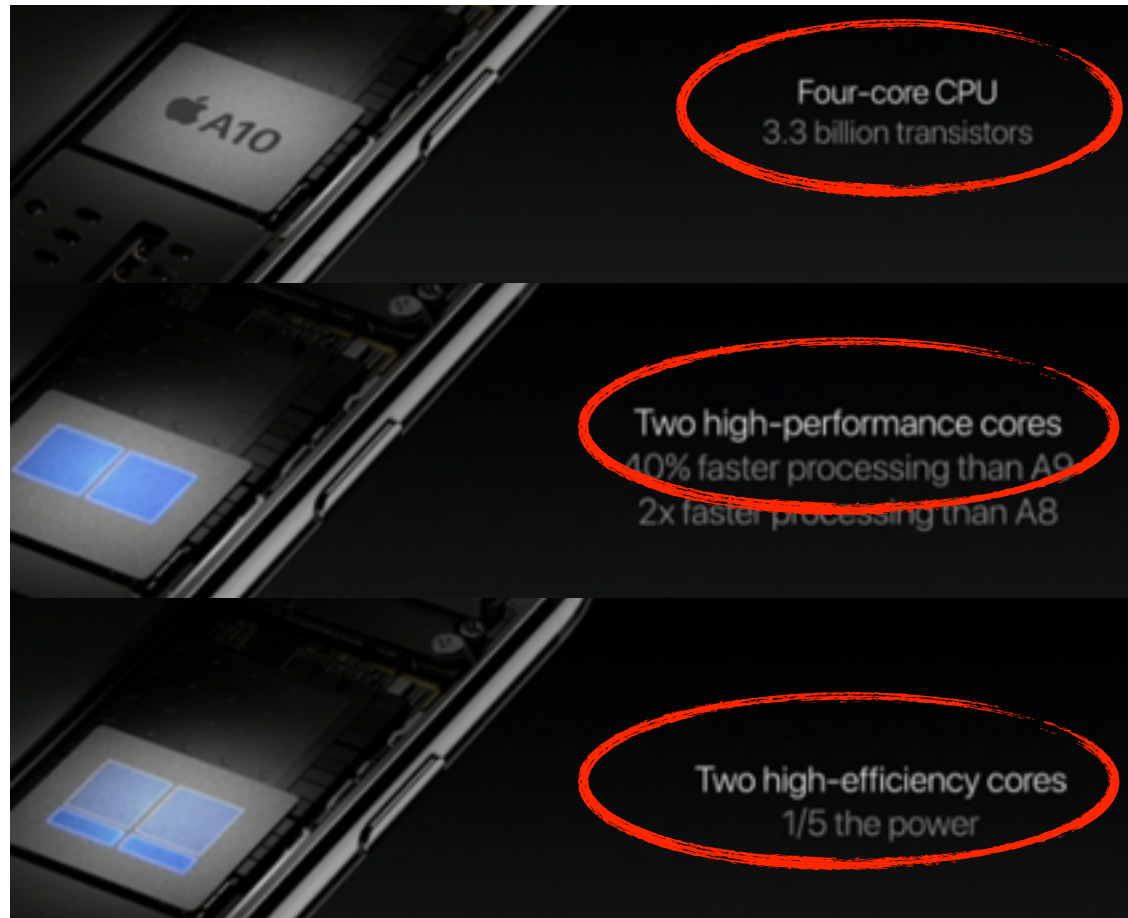
# Asymmetric Multiprocessor (AMP)

- Offer a large performance-energy trade-off space



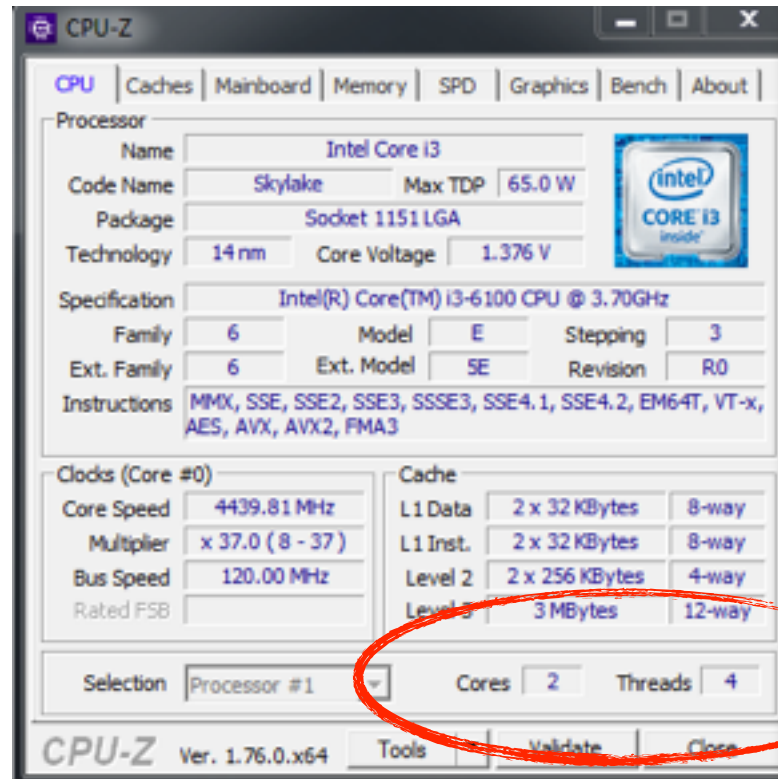
# Asymmetric Chip-Multiprocessor (ACMP)

- Already used in commodity devices (e.g., Samsung Galaxy S6, iPhone 7)



# Combine Multi-core with Hyper-threading

- Common for laptop/desktop/server machine. E.g., 2 physical cores, each core has 2 hyper-threads => 4 virtual cores.
- Not for mobile processors (Hyper-threading costly to implement)



# Today

- Shared variables in multi-threaded programming
  - Mutual exclusion using semaphore
  - Deadlock
- Thread-level parallelism
  - Amdahl's Law: performance model of parallel programs
- **Hardware support for multi-threading**
  - Single-core
  - Hyper-threading
  - Multi-core
  - Cache coherence

# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.



# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.

# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.

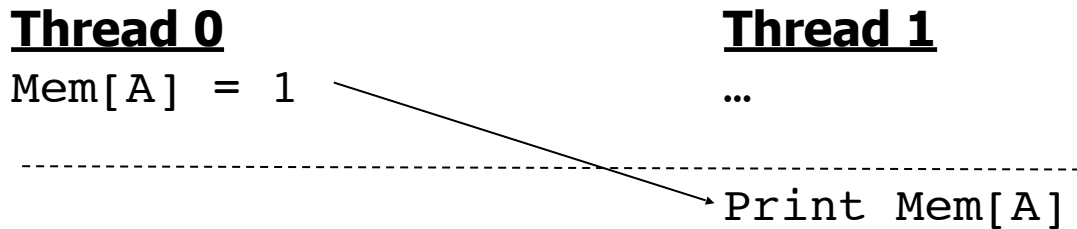
## Thread 0

Mem[A] = 1

## Thread 1

...

Print Mem[A]



The diagram illustrates a race condition in a multi-core processor. It shows two threads, Thread 0 and Thread 1, separated by a horizontal dashed line. Thread 0, on the left, performs the operation 'Mem[A] = 1'. Thread 1, on the right, performs the operation 'Print Mem[A]'. An arrow points from the assignment in Thread 0 to the print statement in Thread 1, indicating the flow of data or the sequence of operations. The dashed line represents a point in time or a barrier between the two threads' actions.

# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone

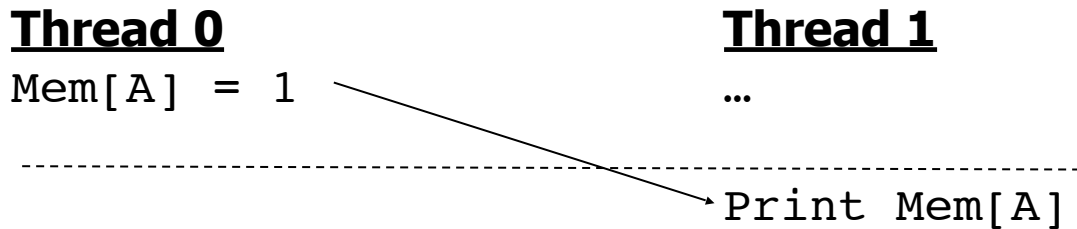
## Thread 0

Mem[A] = 1

## Thread 1

...

Print Mem[A]

A diagram illustrating a race condition. On the left, under the heading 'Thread 0', the code 'Mem[A] = 1' is shown. On the right, under the heading 'Thread 1', there is an ellipsis '...' followed by 'Print Mem[A]'. A horizontal dashed line separates the two threads. An arrow points from the assignment 'Mem[A] = 1' in Thread 0 to the 'Print Mem[A]' statement in Thread 1, crossing the dashed line to indicate that Thread 1's print statement occurs after Thread 0's write.

# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone
- **Basic question:** If multiple cores access the same data, how do they ensure they all see a consistent state?

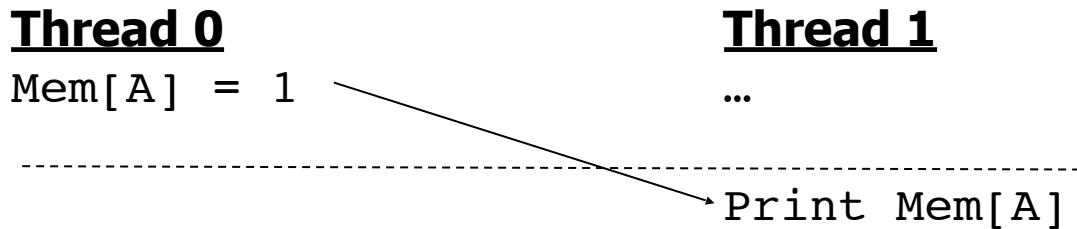
## Thread 0

Mem[A] = 1

## Thread 1

...

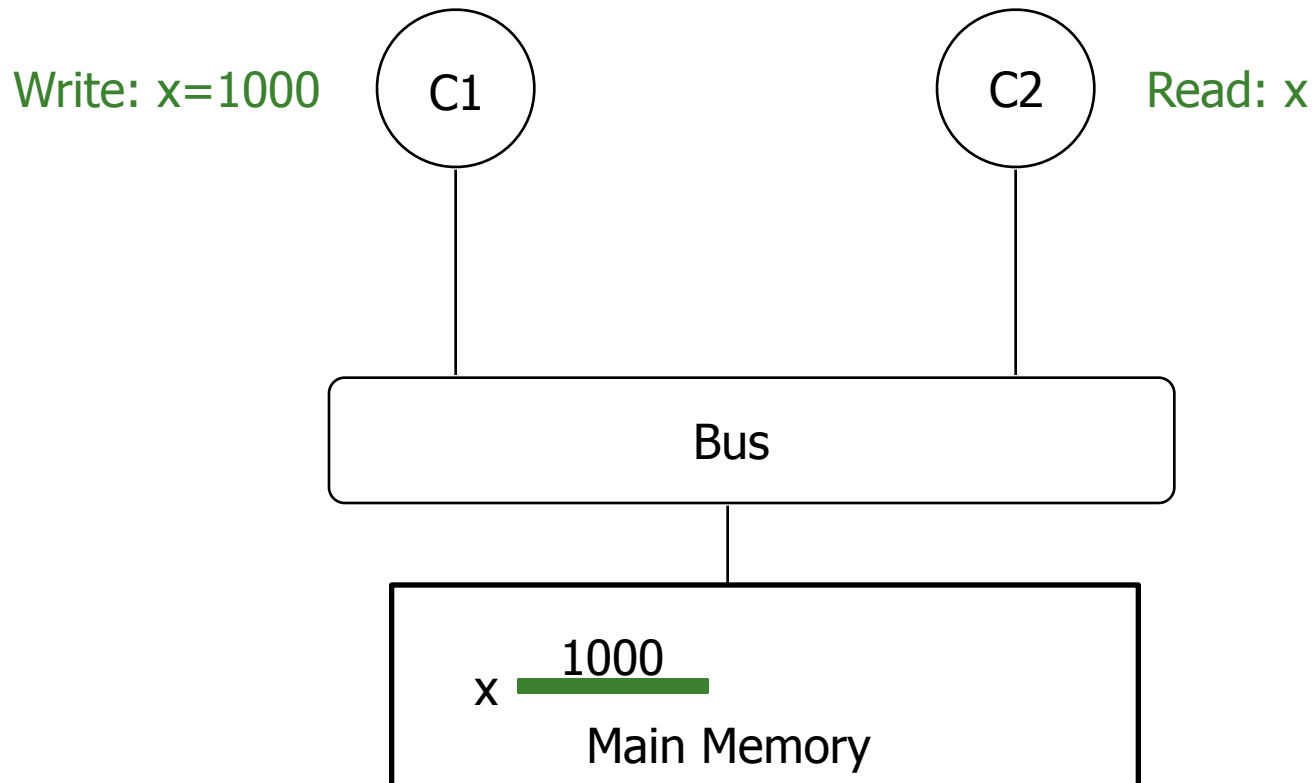
Print Mem[A]



The diagram illustrates a sequence of operations across two threads. On the left, under the heading 'Thread 0', the operation 'Mem[A] = 1' is shown. On the right, under the heading 'Thread 1', an ellipsis '...' is shown above the operation 'Print Mem[A]'. A horizontal dashed line spans the width of the diagram, representing a timeline or sequence of events. An arrow originates from the text 'Mem[A] = 1' and points diagonally down and to the right, crossing the dashed line and ending at the text 'Print Mem[A]'. This visualizes Thread 1's print operation occurring after Thread 0's write operation.

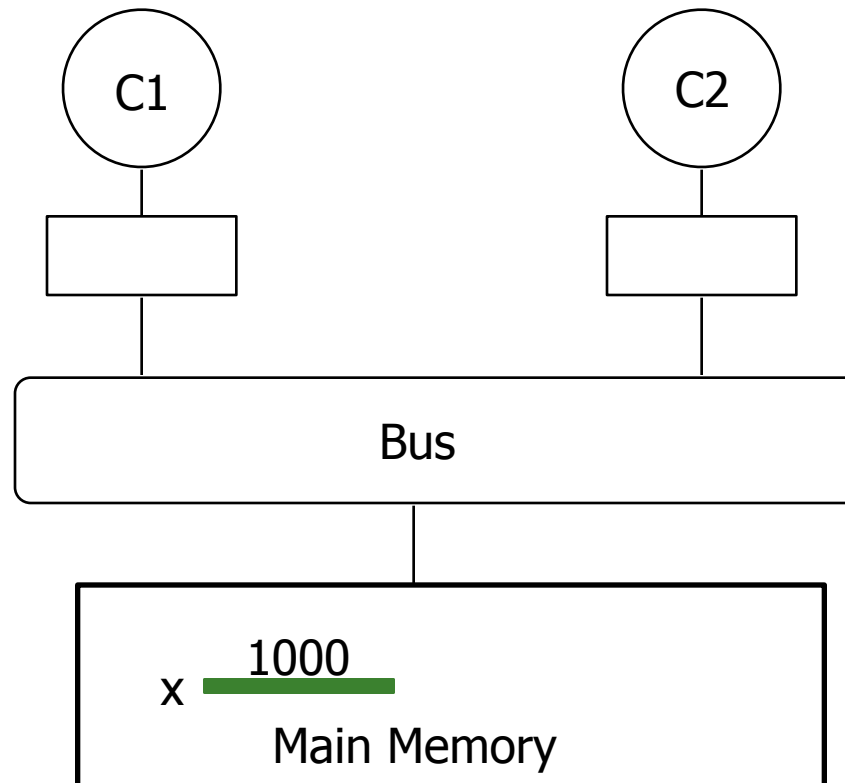
# The Issue

- Without cache, the issue is (theoretically) solvable by using mutex.
- ...because there is only one copy of  $x$  in the entire system. Accesses to  $x$  in memory are serialized by mutex.



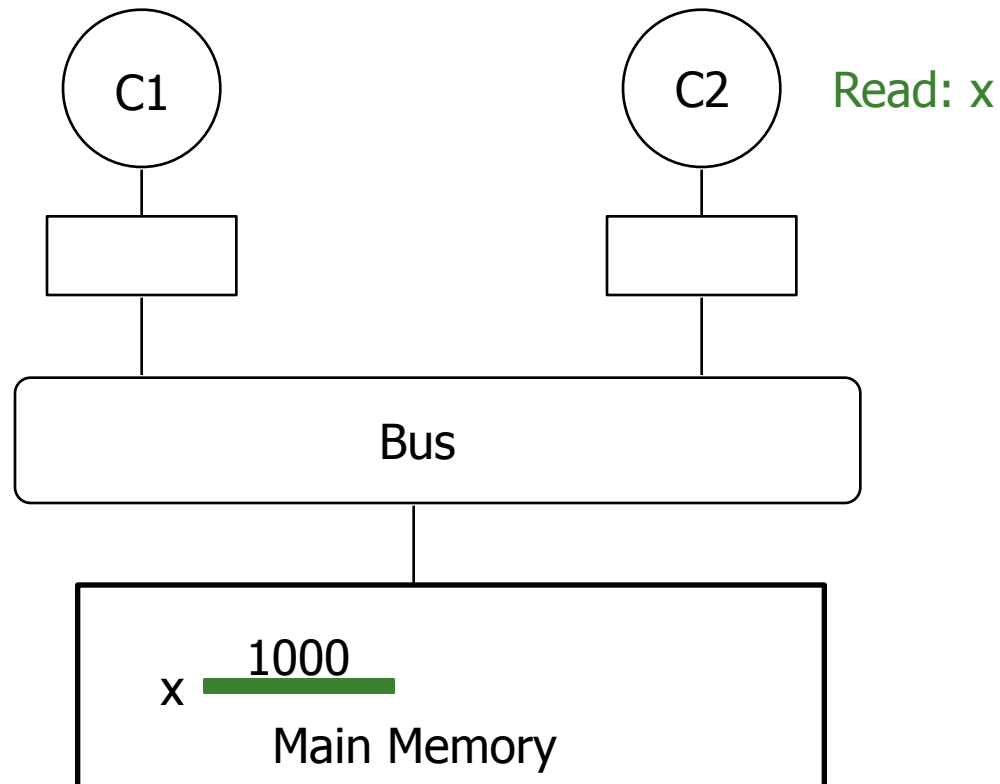
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



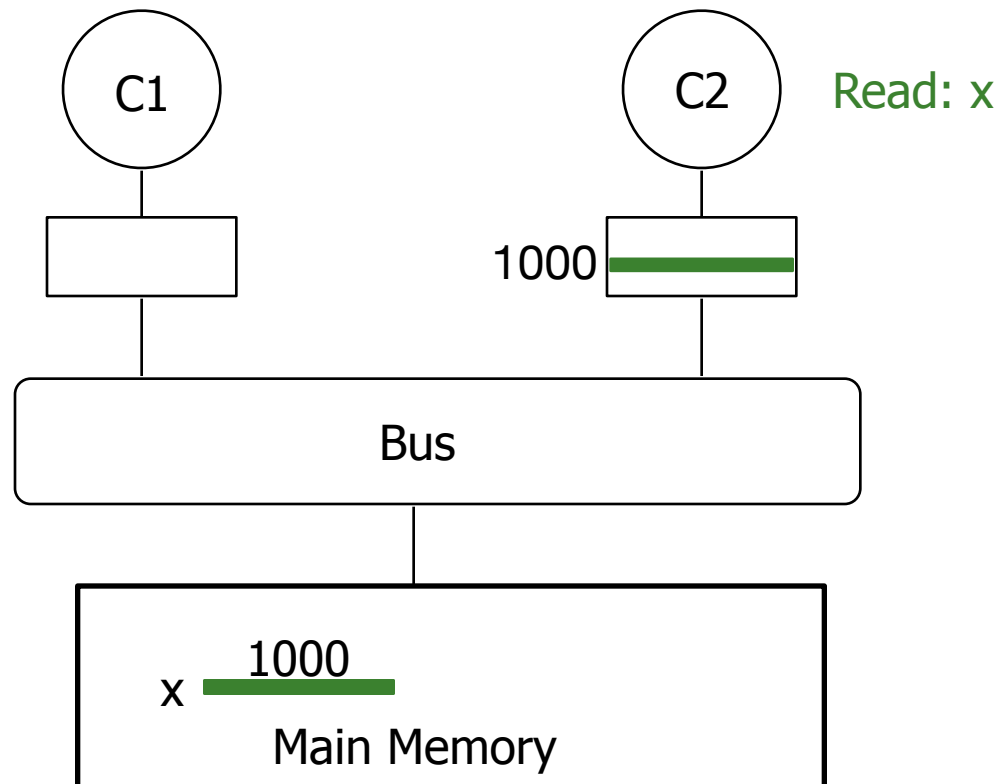
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



# The Issue

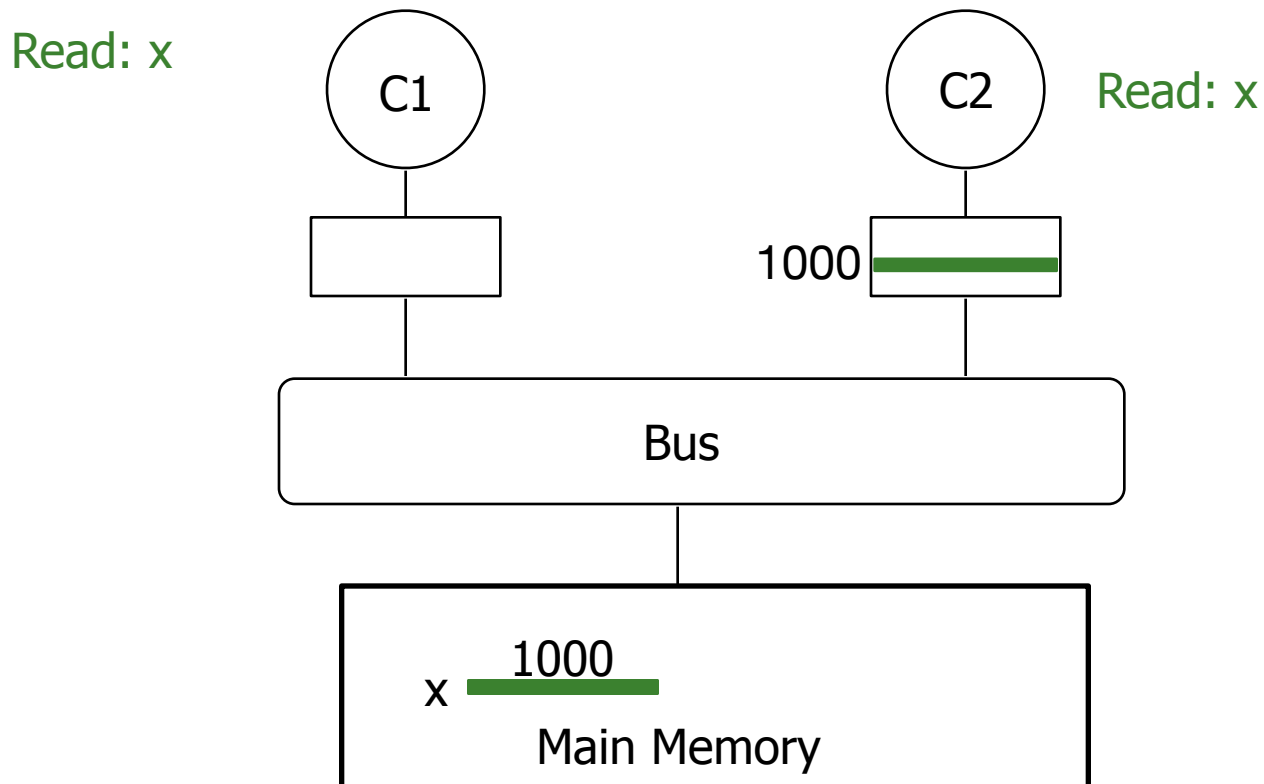
- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)





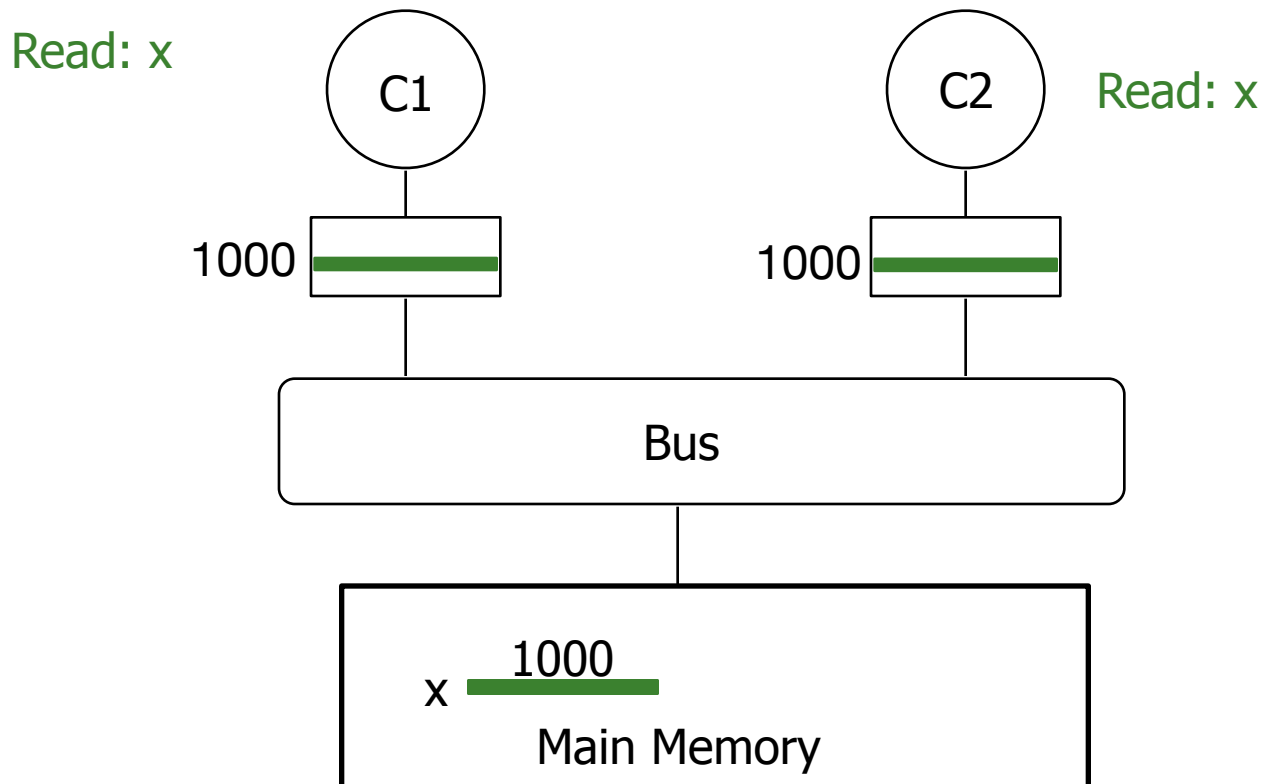
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



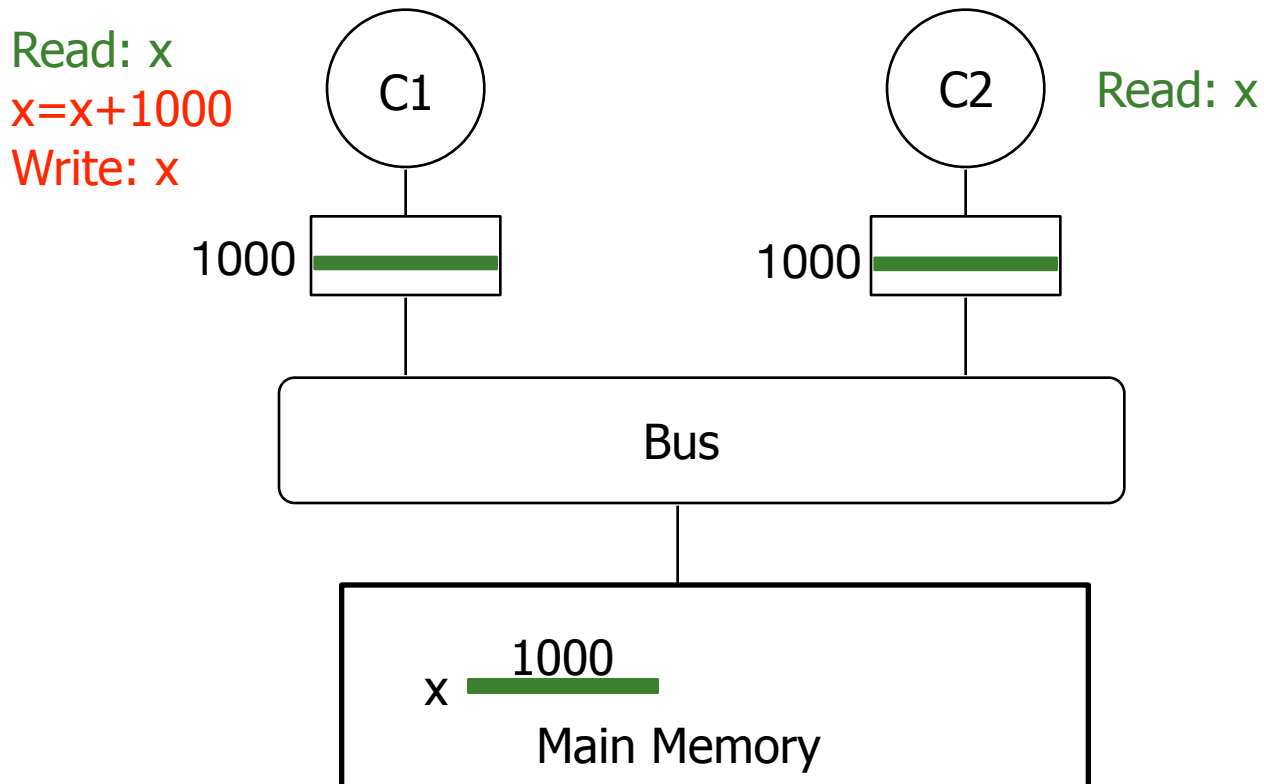
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



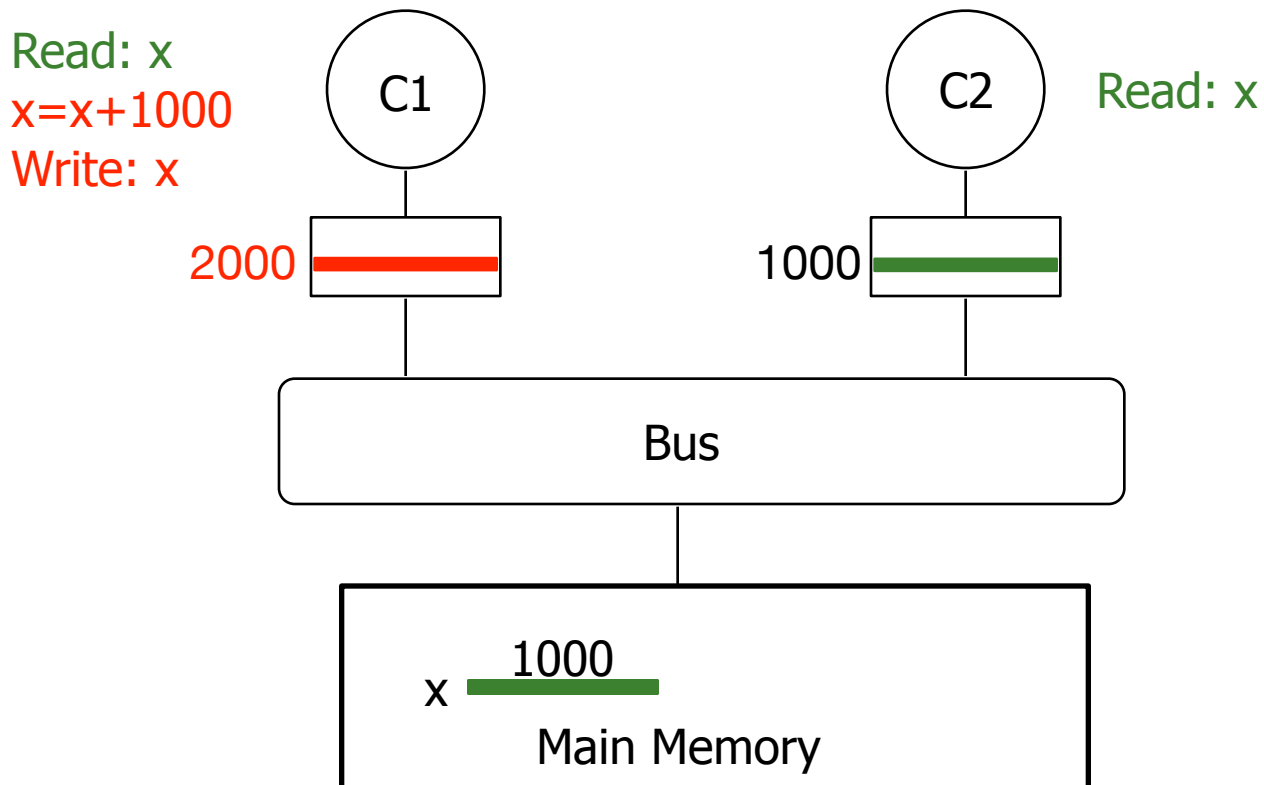
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



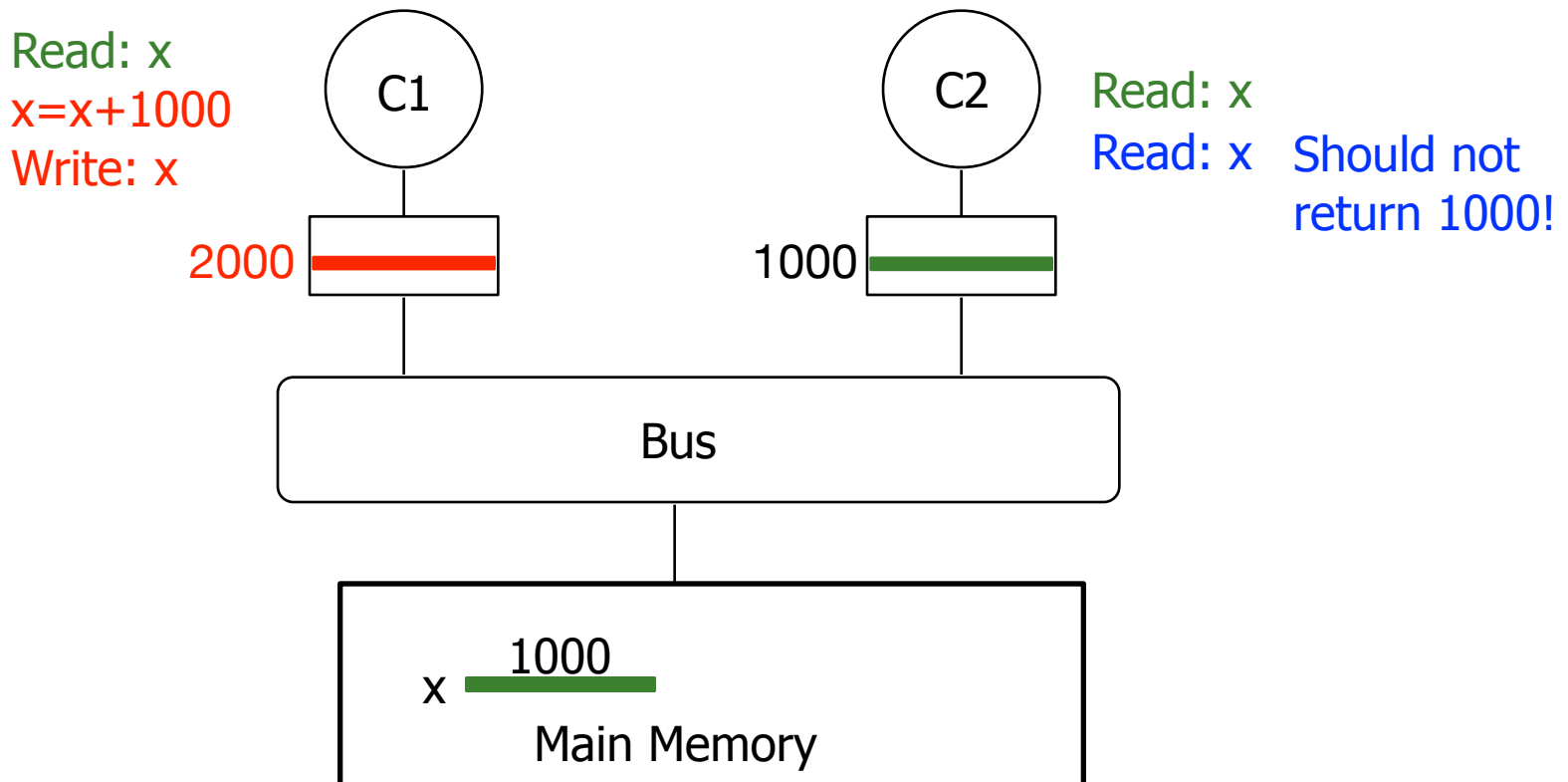
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



# Cache Coherence: The Idea

- **Key issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.

# Cache Coherence: The Idea

- **Key issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Key idea:** ensure multiple copies have same value, i.e., *coherent*

# Cache Coherence: The Idea

- **Key issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Key idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:



# Cache Coherence: The Idea

- **Key issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Key idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
  - **Update:** push new value to all copies (in other caches)

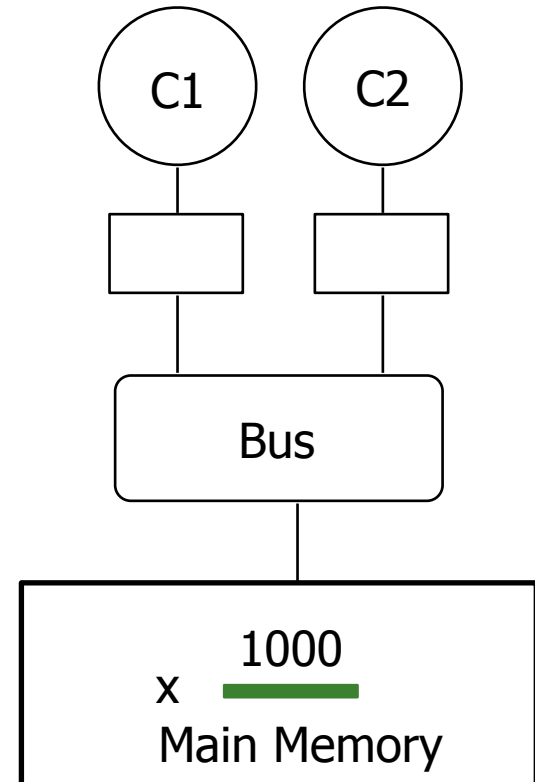
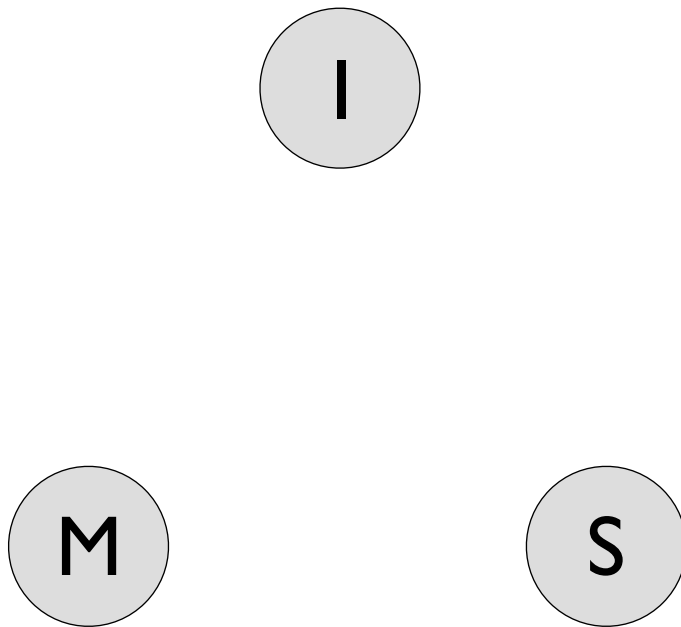
# Cache Coherence: The Idea

- **Key issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Key idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
  - **Update:** push new value to all copies (in other caches)
  - **Invalidate:** invalidate other copies (in other caches)

# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

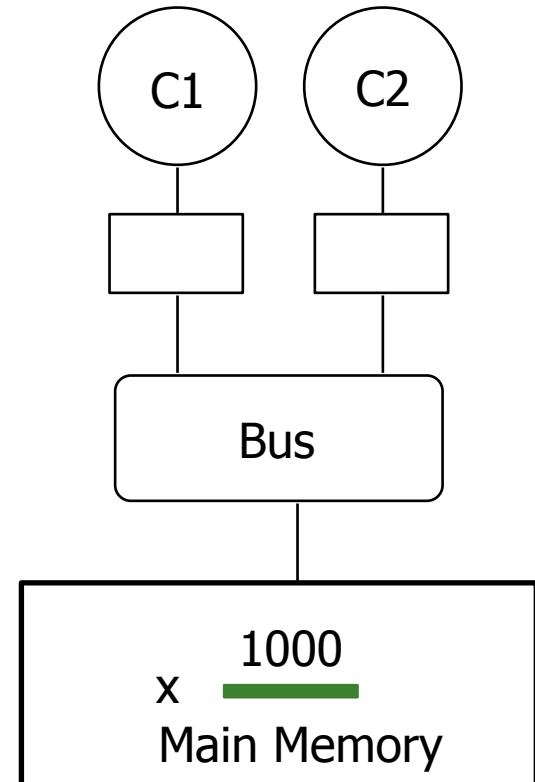
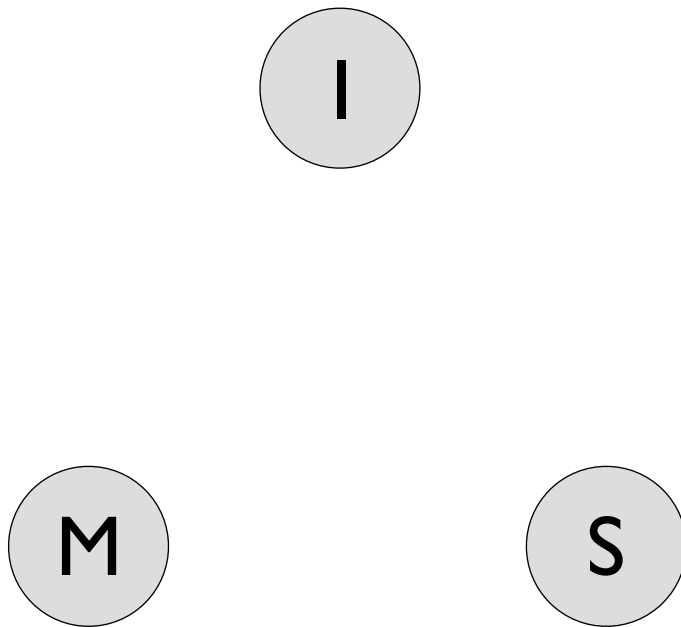


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Read:  $x$

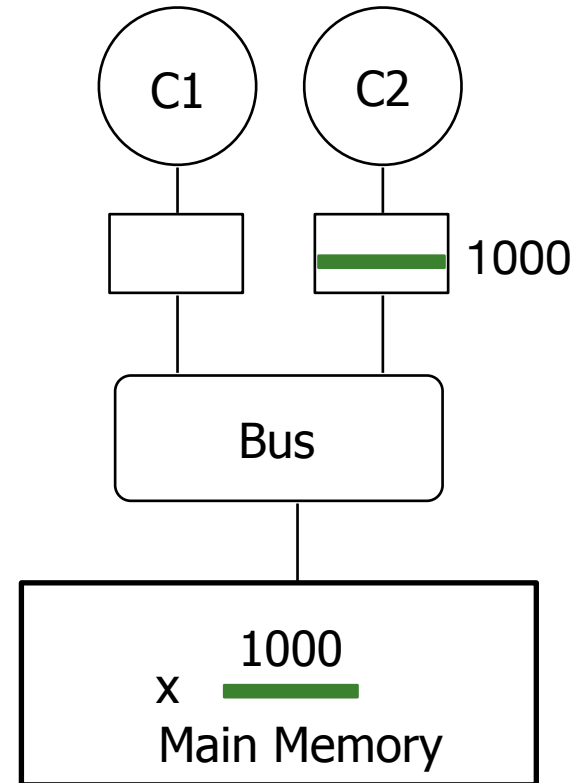
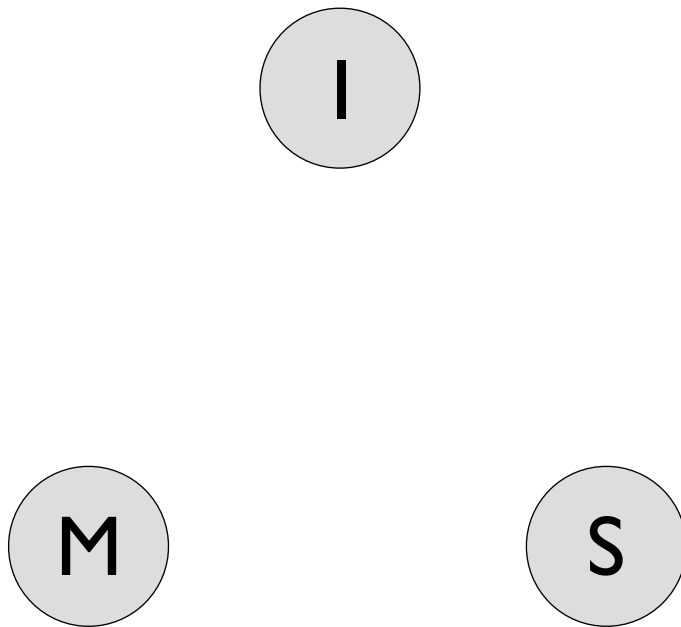


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Read:  $x$

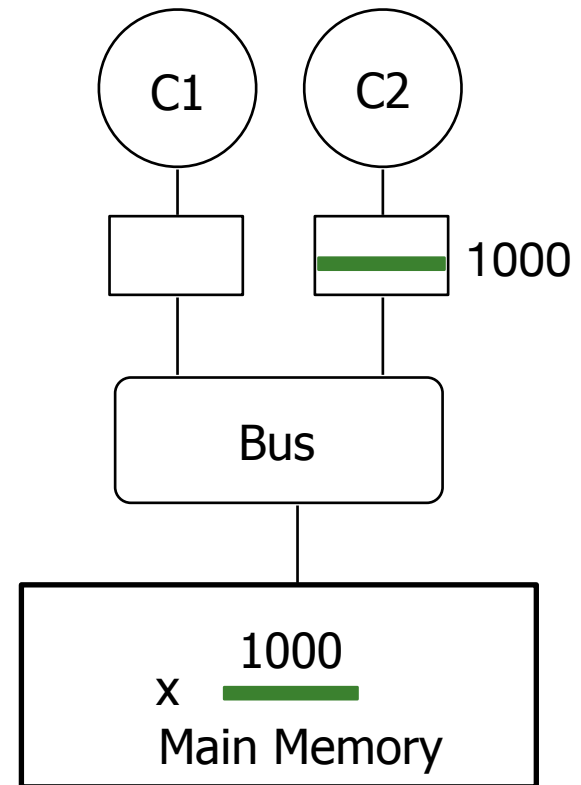
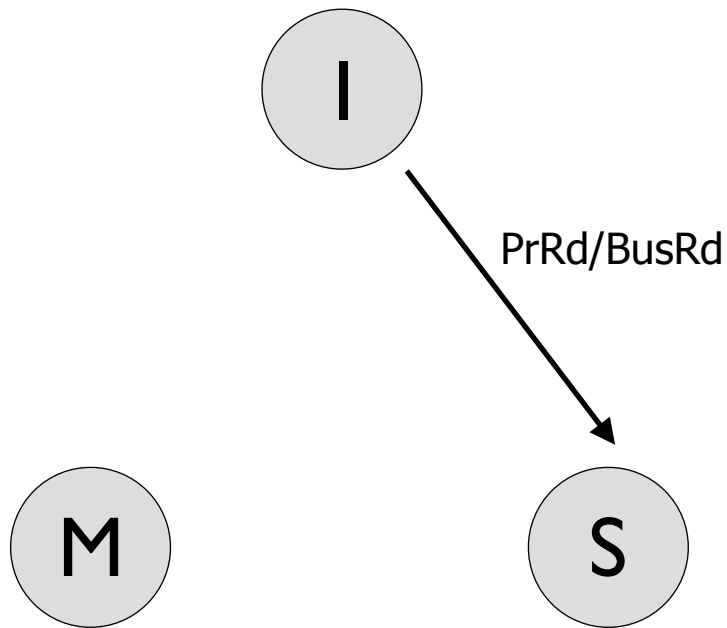


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

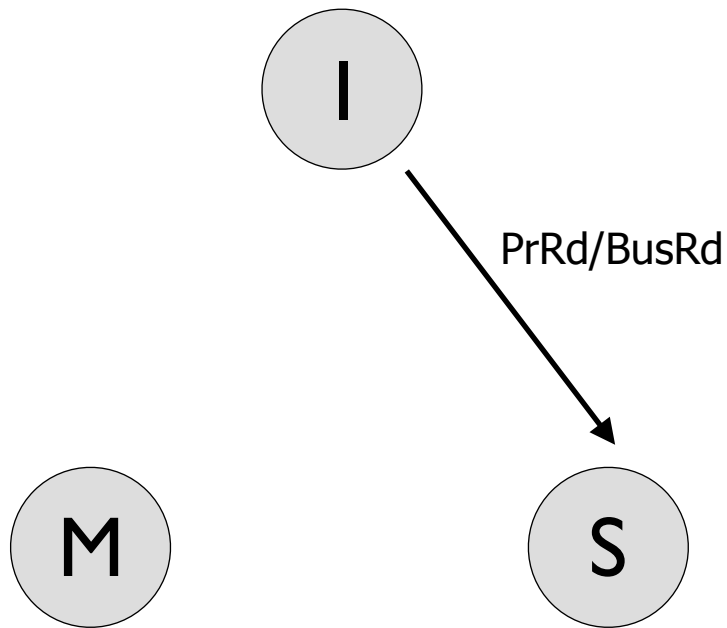
Read:  $x$



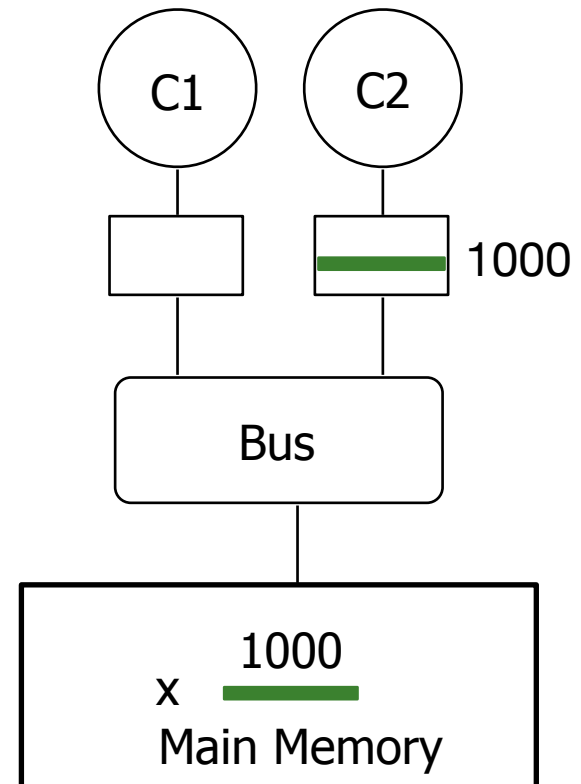
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



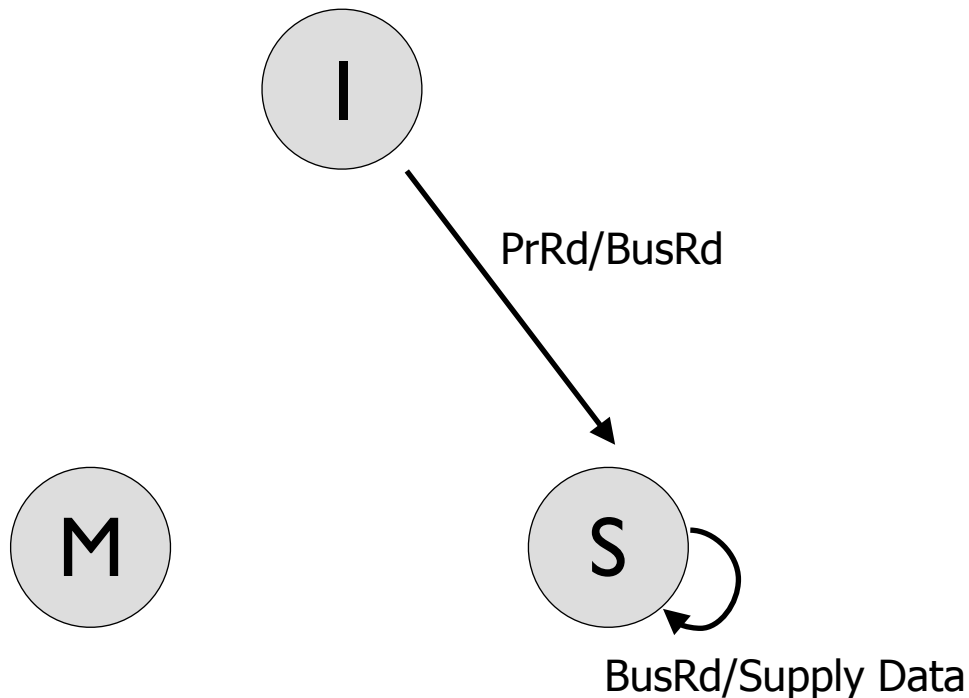
Read:  $x$    Read:  $x$



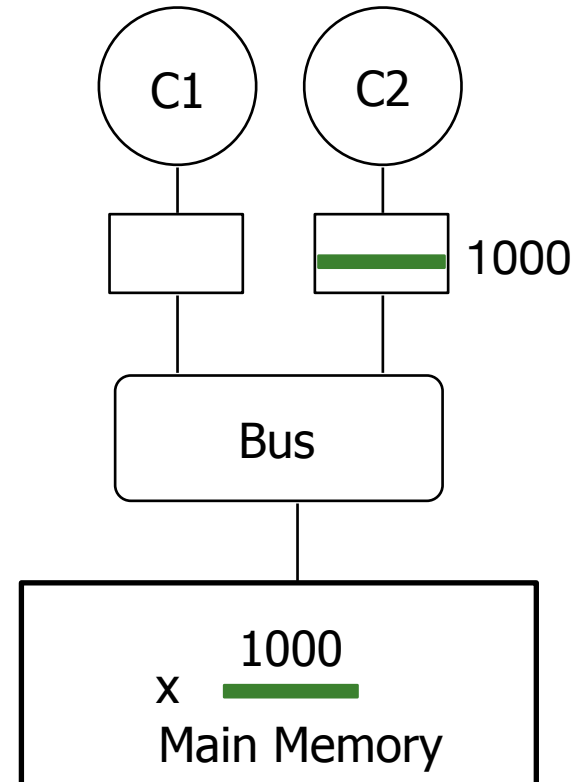
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



Read:  $x$    Read:  $x$

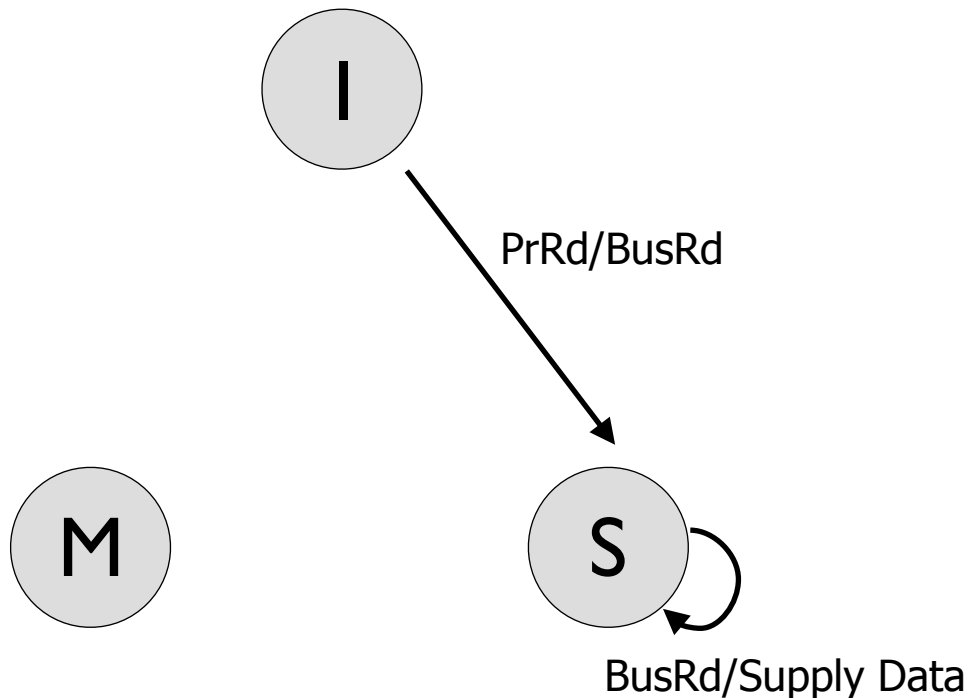




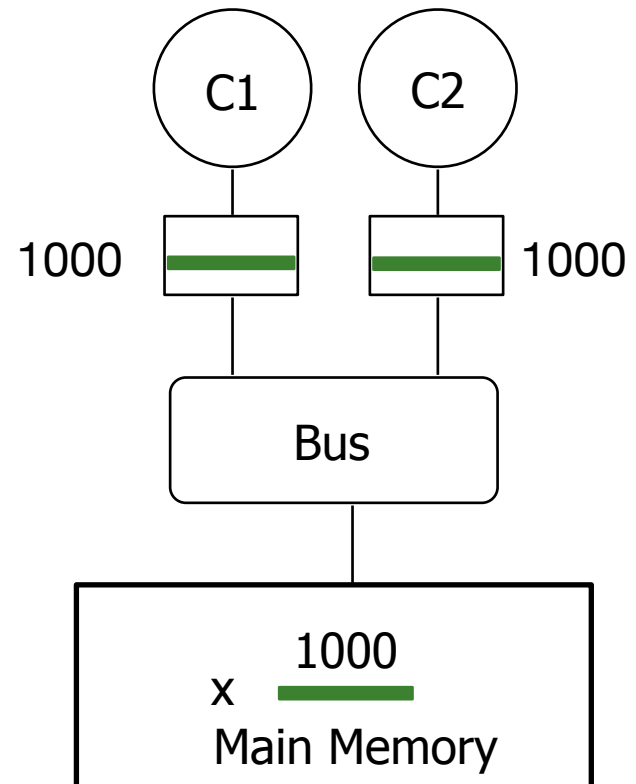
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



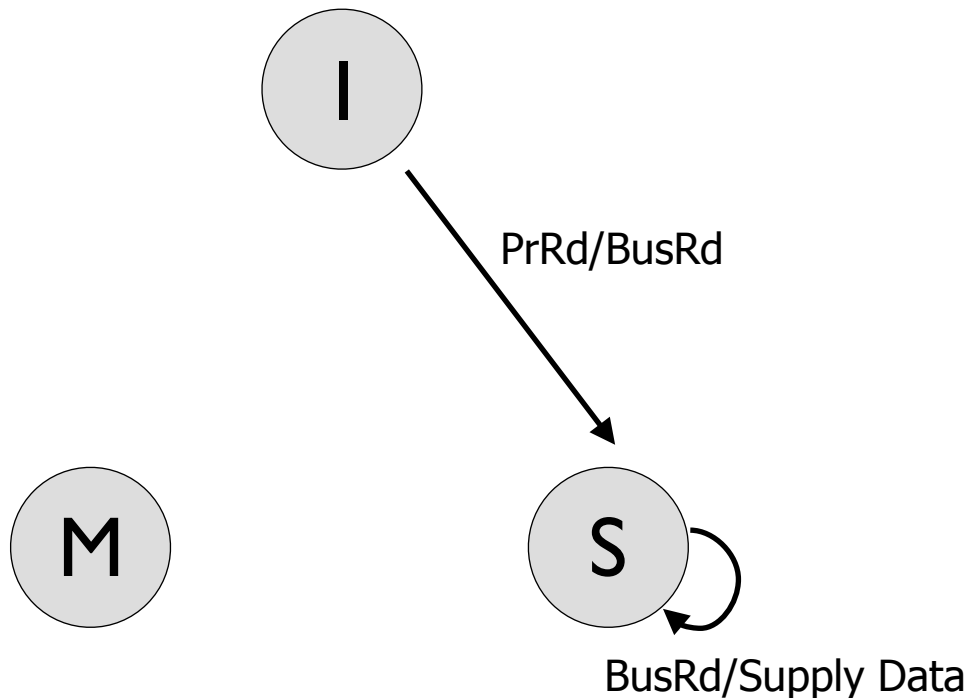
Read:  $x$    Read:  $x$



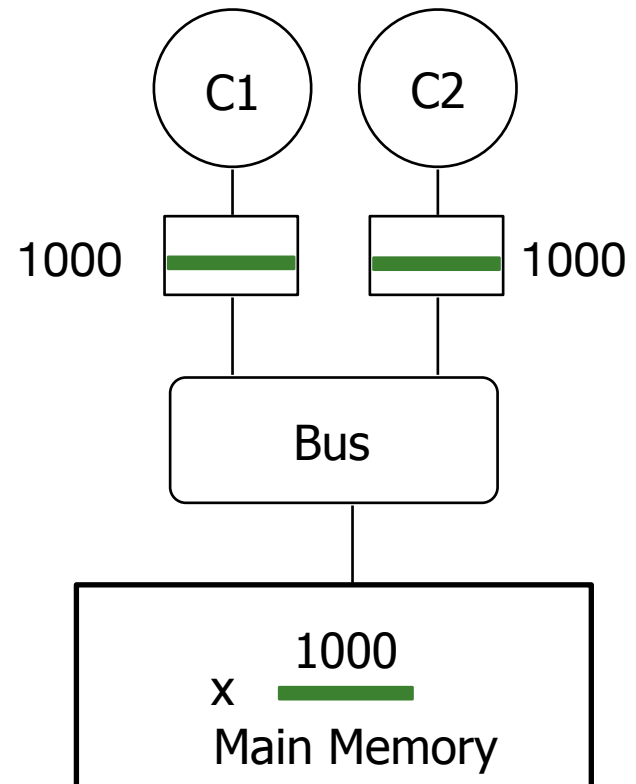
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;  
Syntax: Event/Action



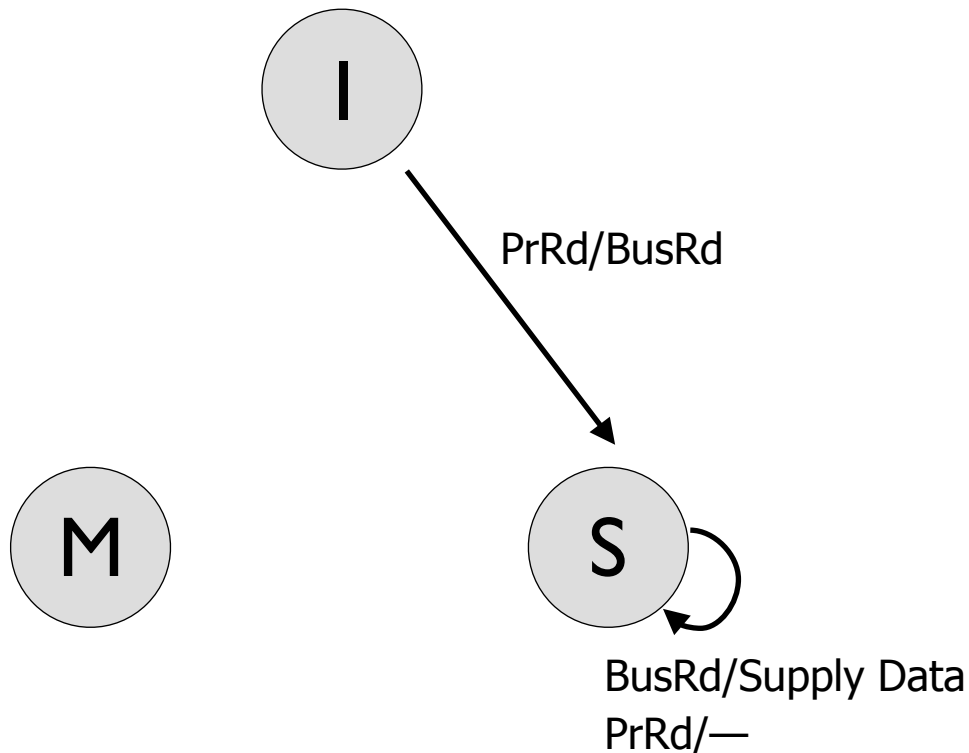
Read: x    Read: x  
              Read: x



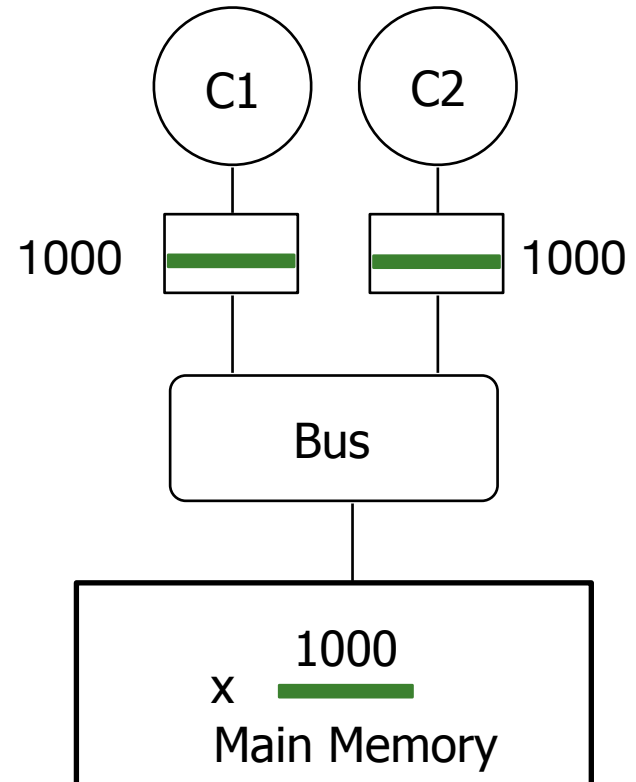
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;  
Syntax: Event/Action



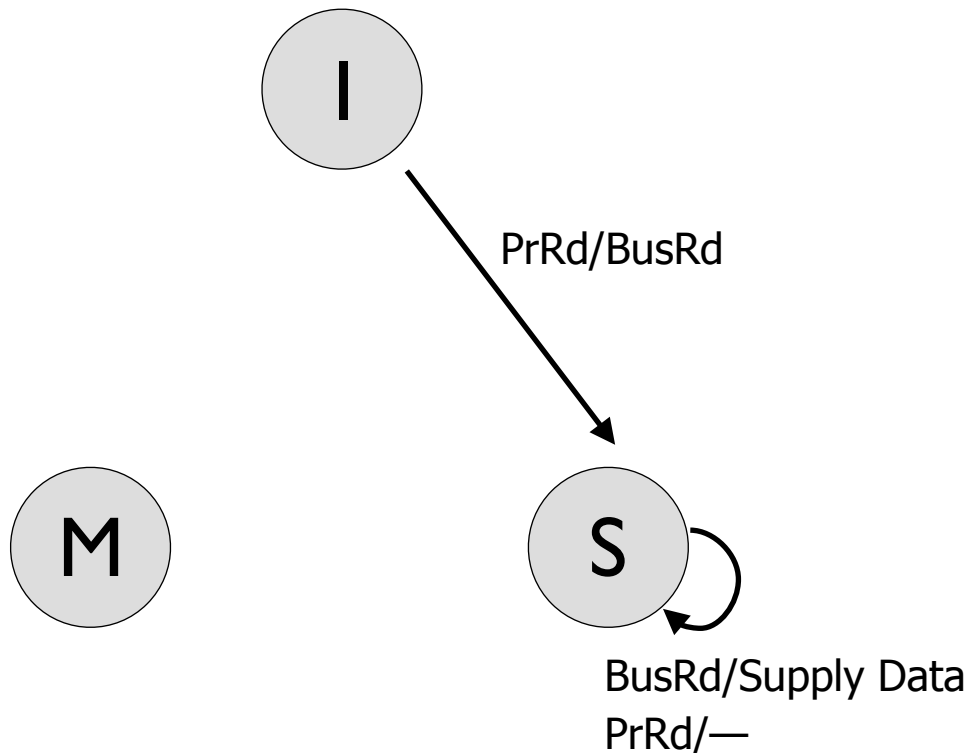
Read: x    Read: x  
              Read: x



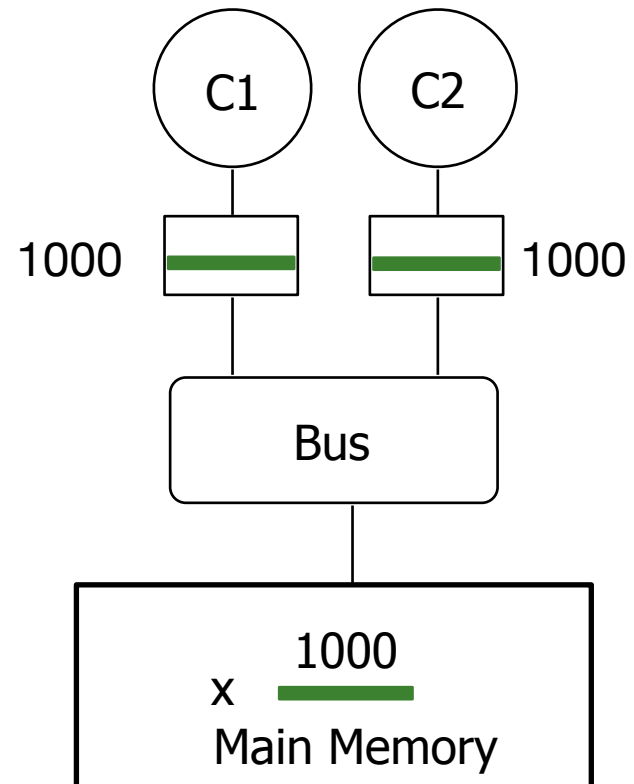
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



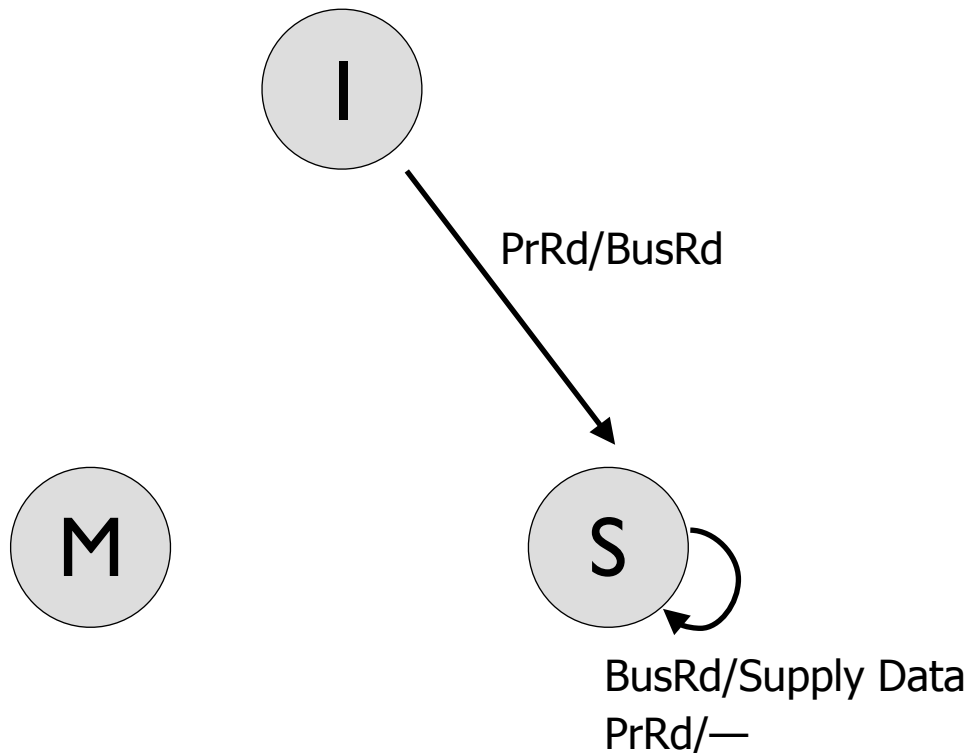
Read:  $x$     Read:  $x$   
Read:  $x$   
Write:  $x = 5000$



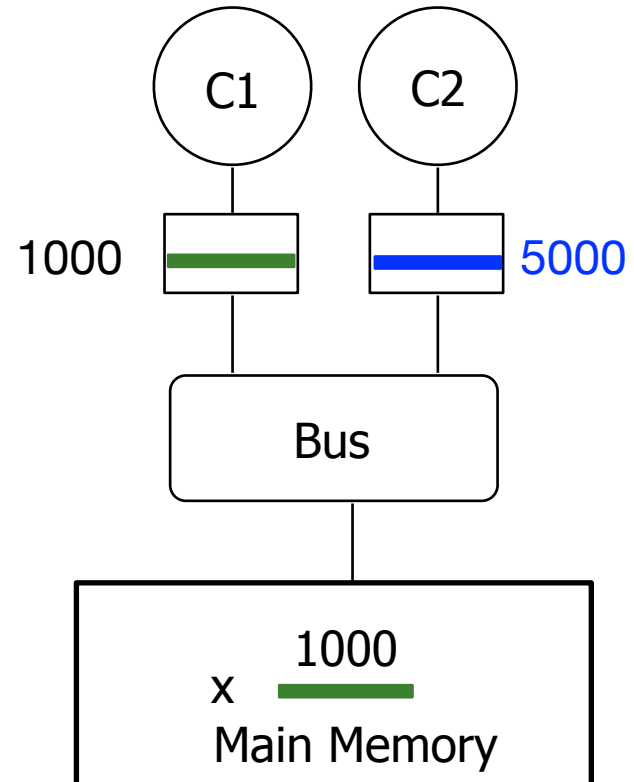
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



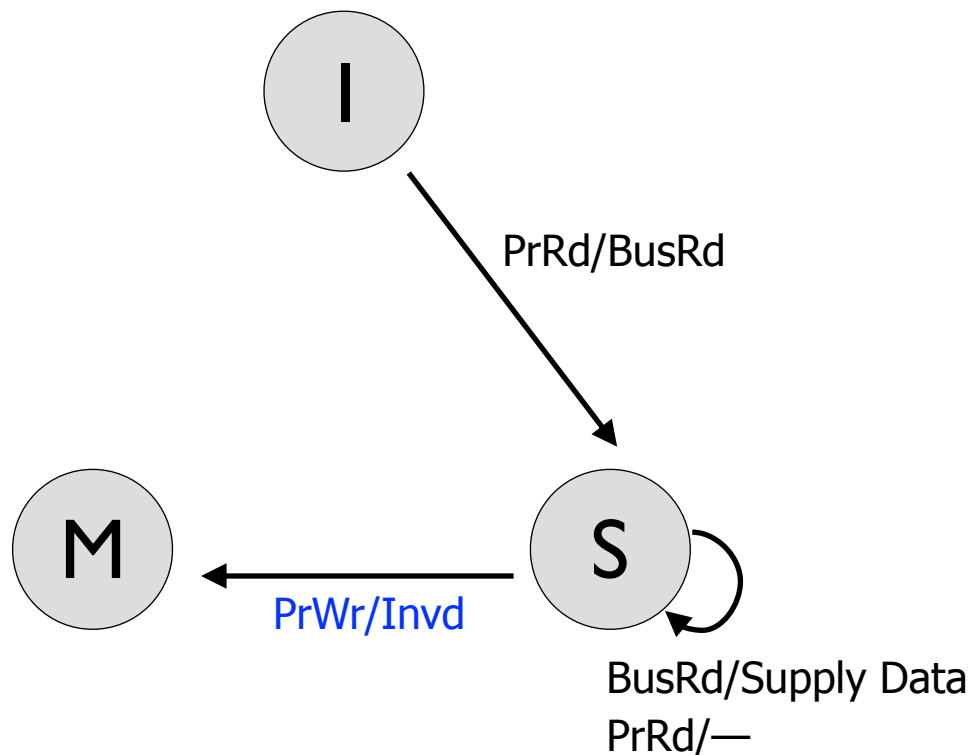
Read:  $x$     Read:  $x$   
Read:  $x$   
Write:  $x = 5000$



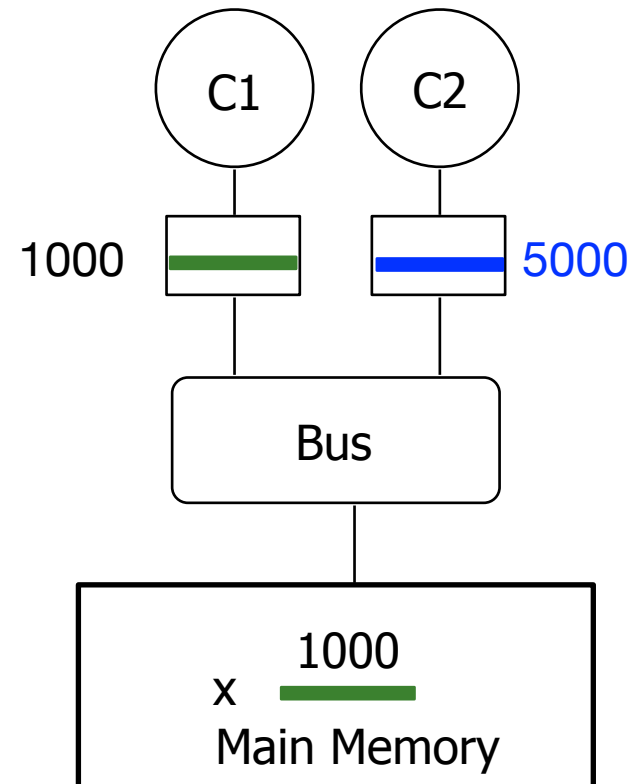
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



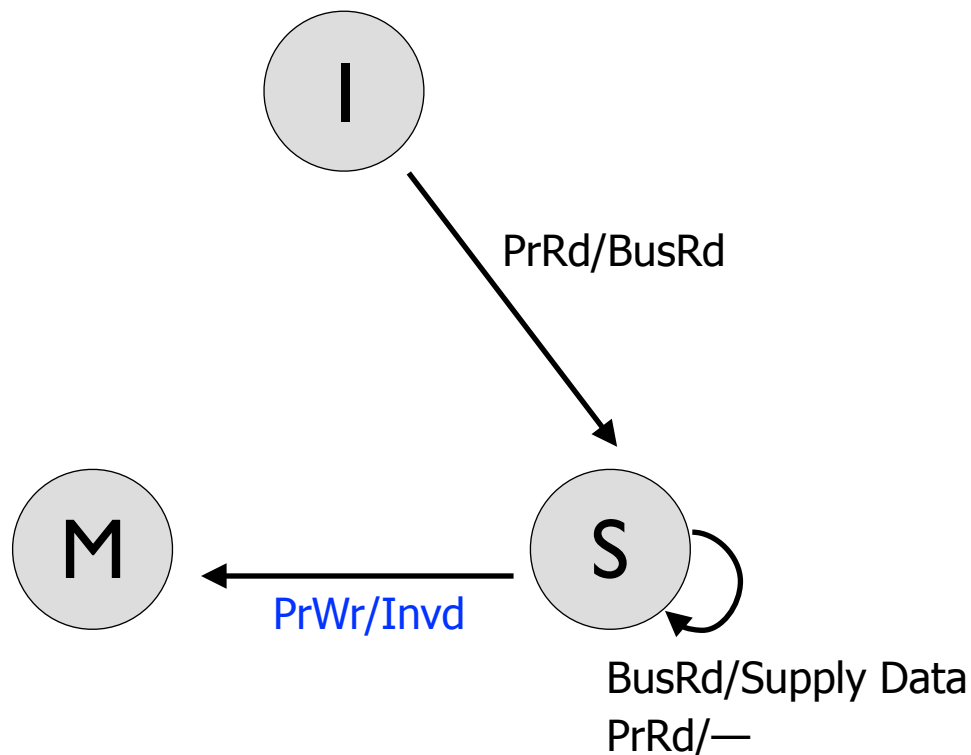
Read:  $x$     Read:  $x$   
Read:  $x$   
Write:  $x = 5000$



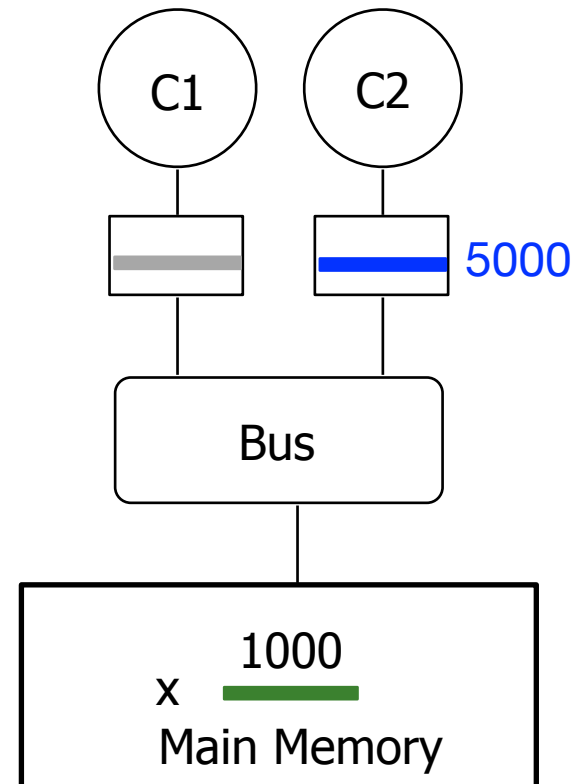
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



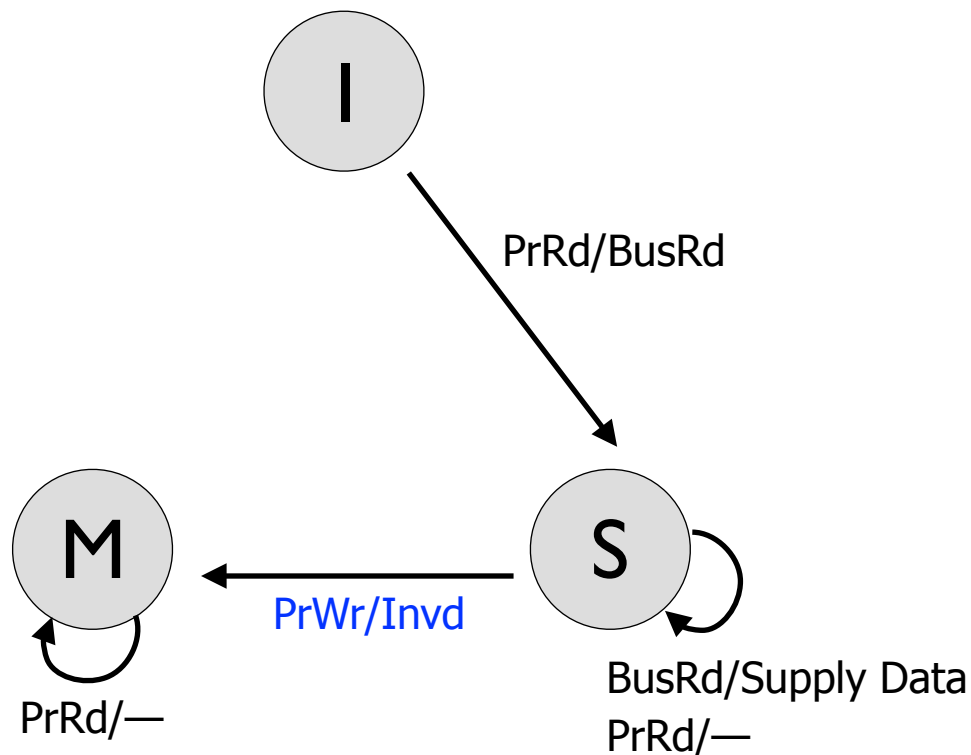
Read:  $x$     Read:  $x$   
Read:  $x$   
Write:  $x = 5000$



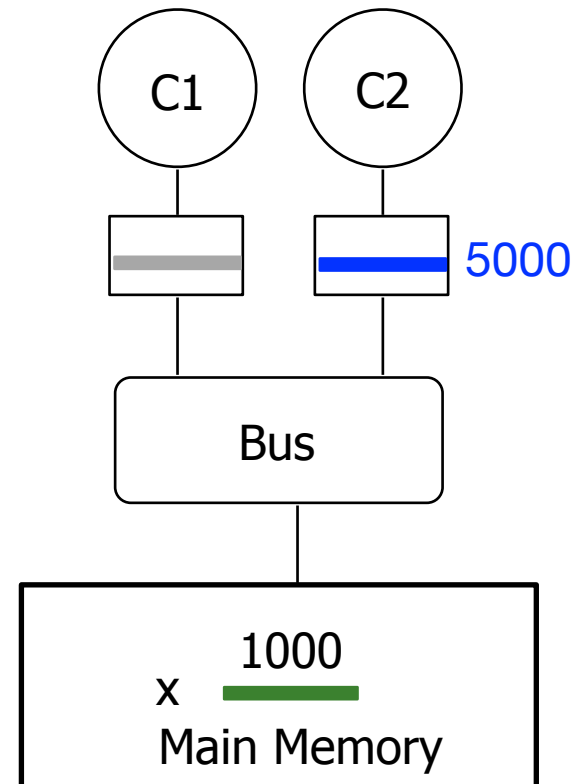
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



Read:  $x$     Read:  $x$   
Read:  $x$   
Write:  $x = 5000$

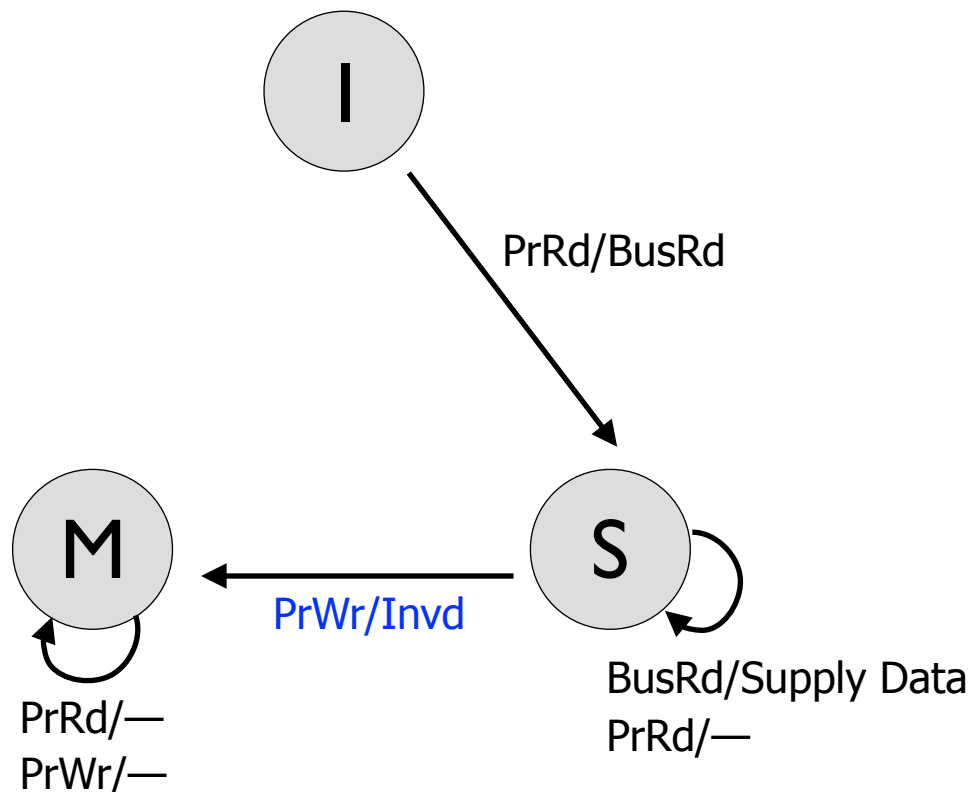




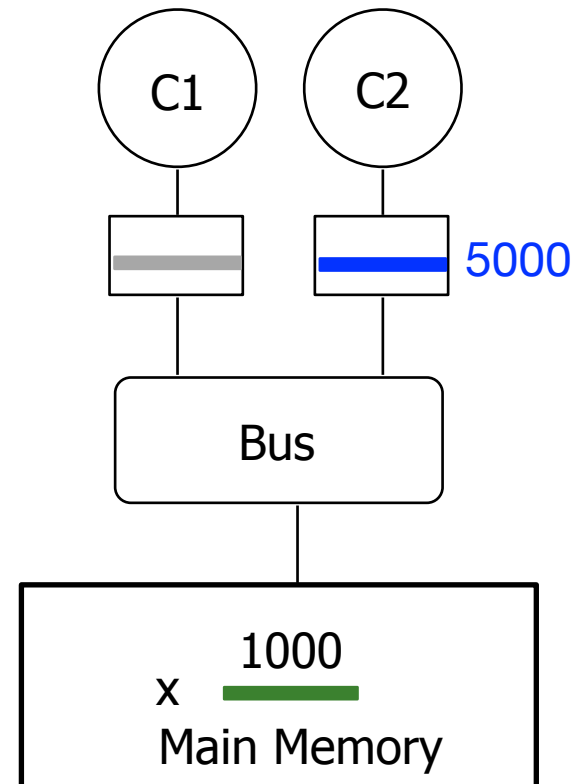
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;  
Syntax: Event/Action



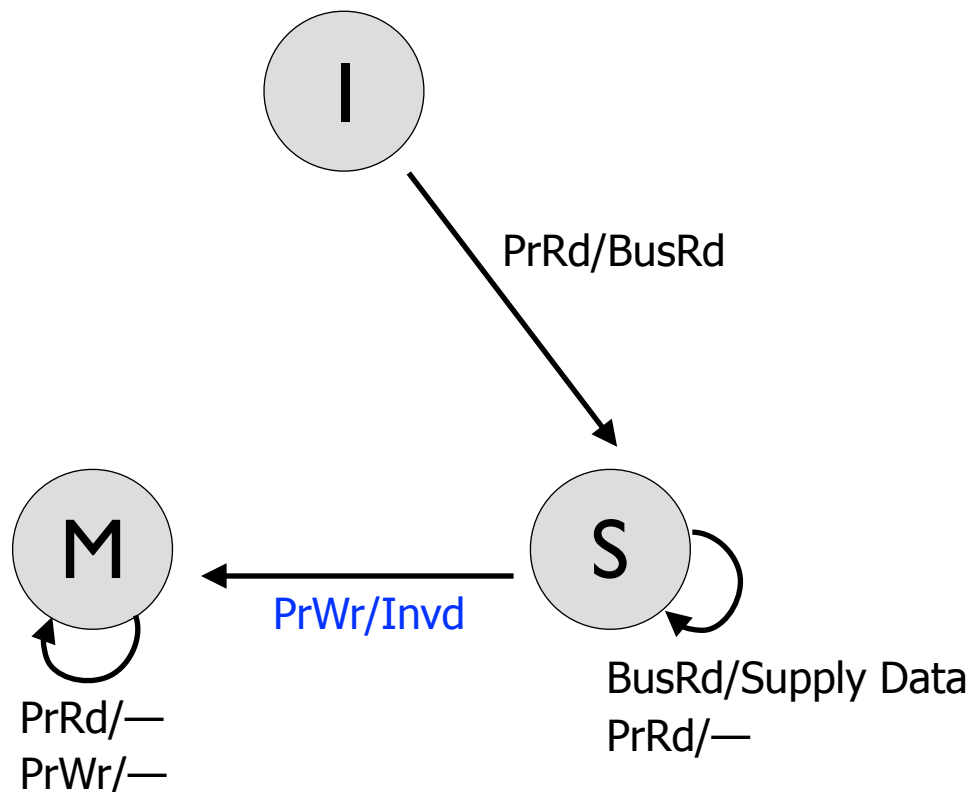
Read: x    Read: x  
Read: x  
Write: x = 5000



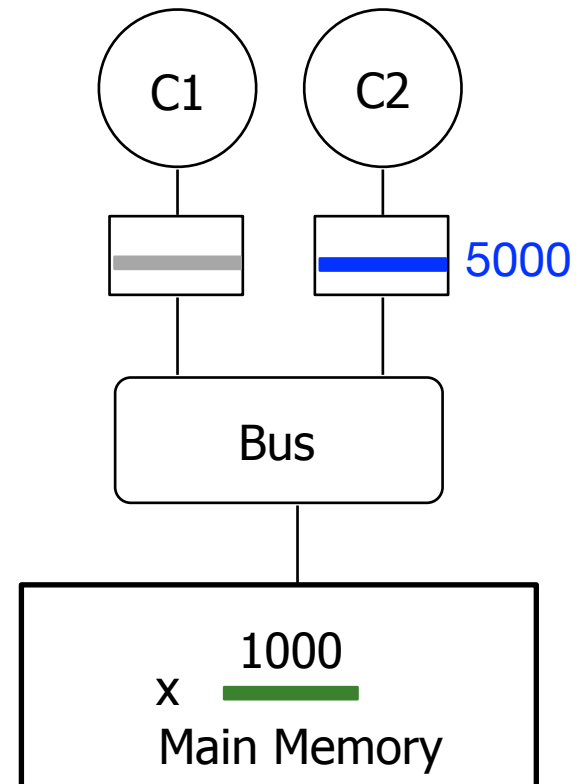
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;  
Syntax: Event/Action



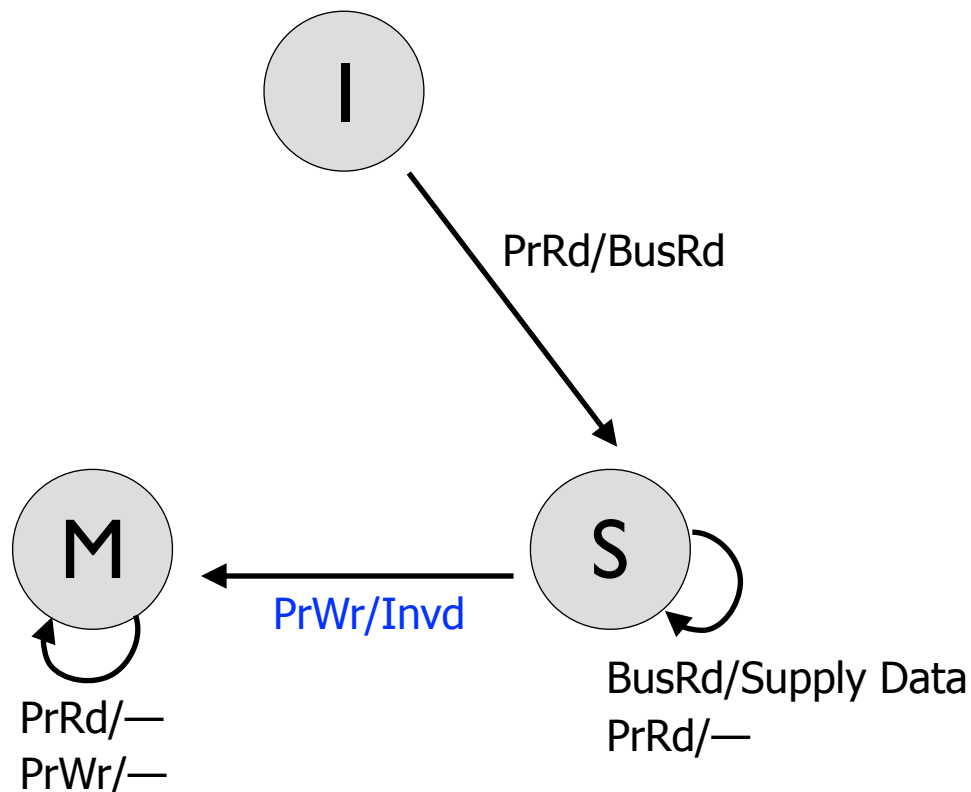
Read: x    Read: x  
Read: x    Read: x  
Write: x = 5000



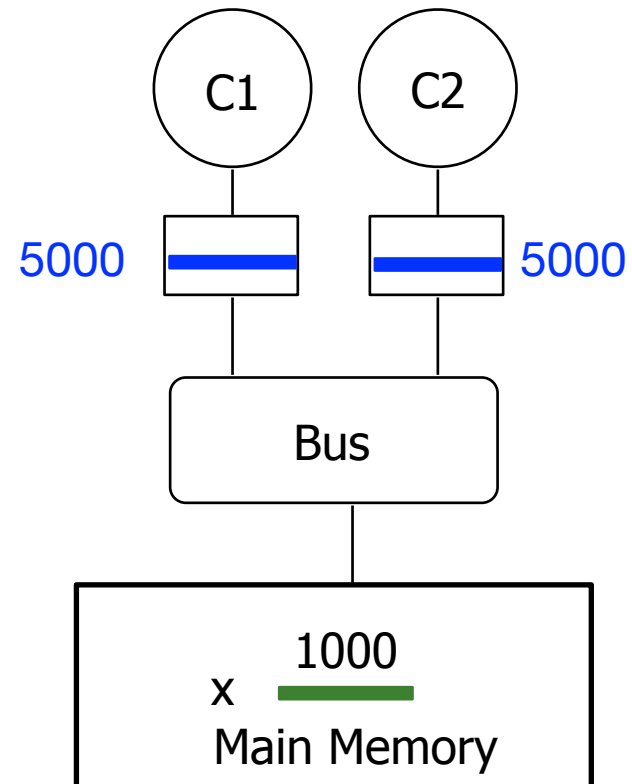
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;  
Syntax: Event/Action



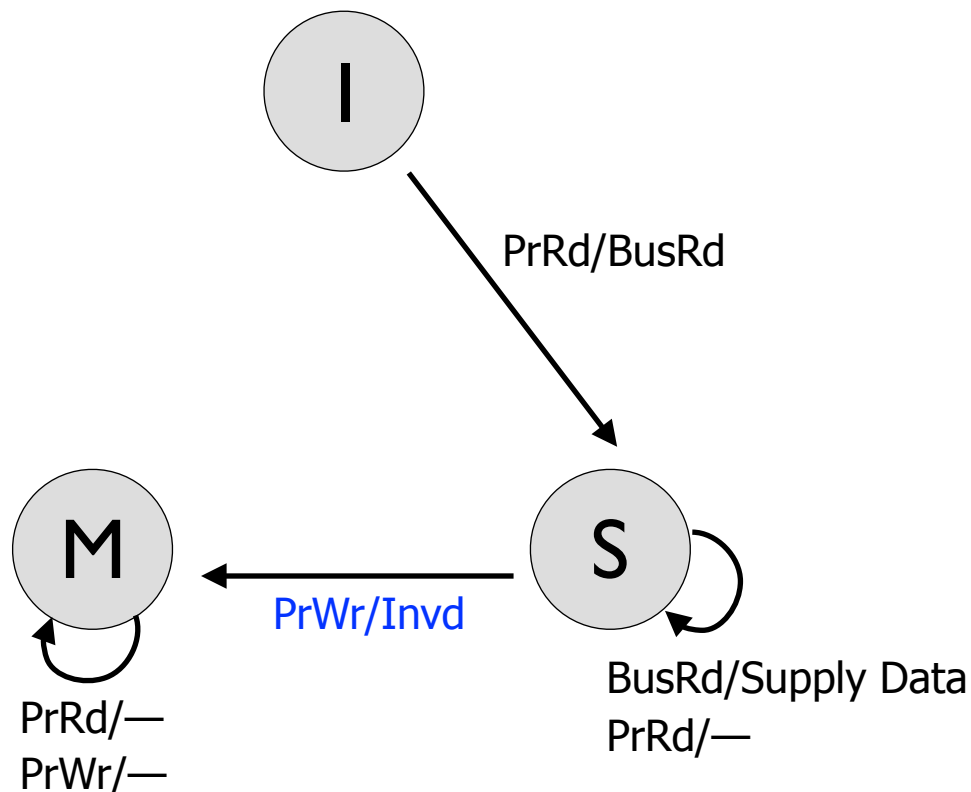
Read: x    Read: x  
Read: x    Read: x  
Write: x = 5000



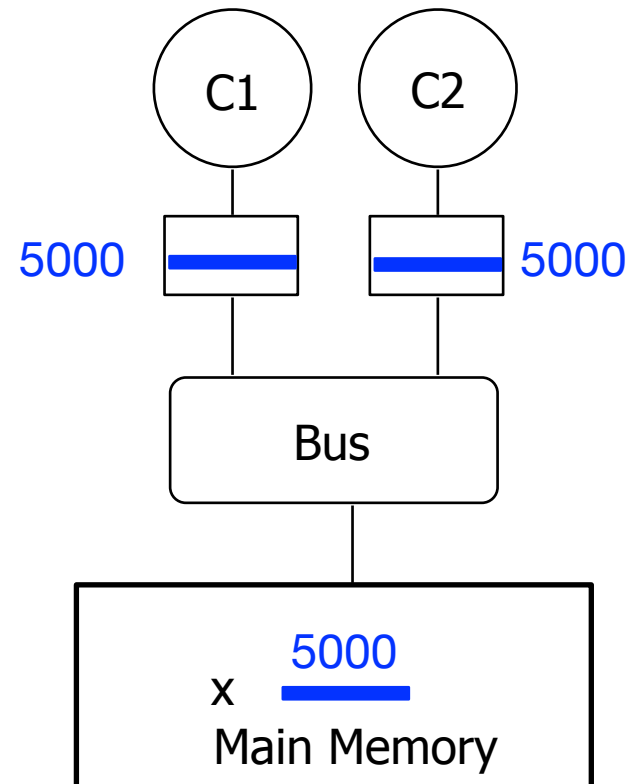
# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;  
Syntax: Event/Action



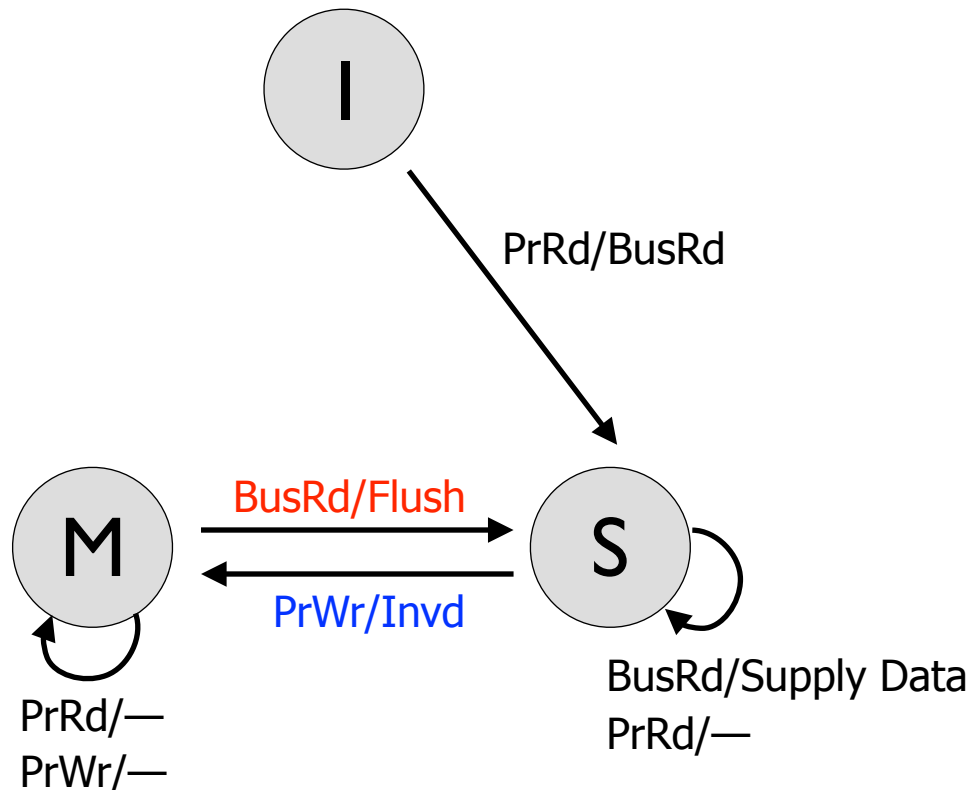
Read: x    Read: x  
Read: x    Read: x  
Write: x = 5000



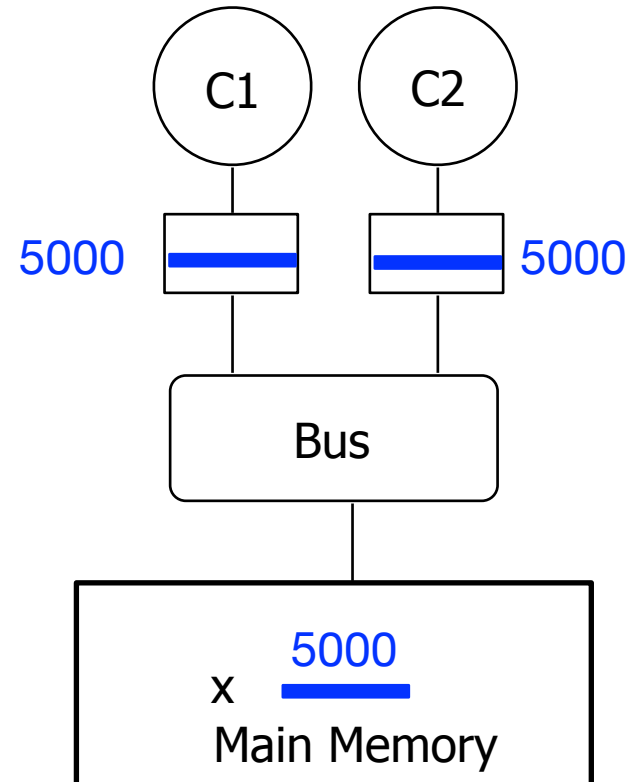
# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action



Read:  $x$     Read:  $x$   
Read:  $x$     Read:  $x$   
Write:  $x = 5000$

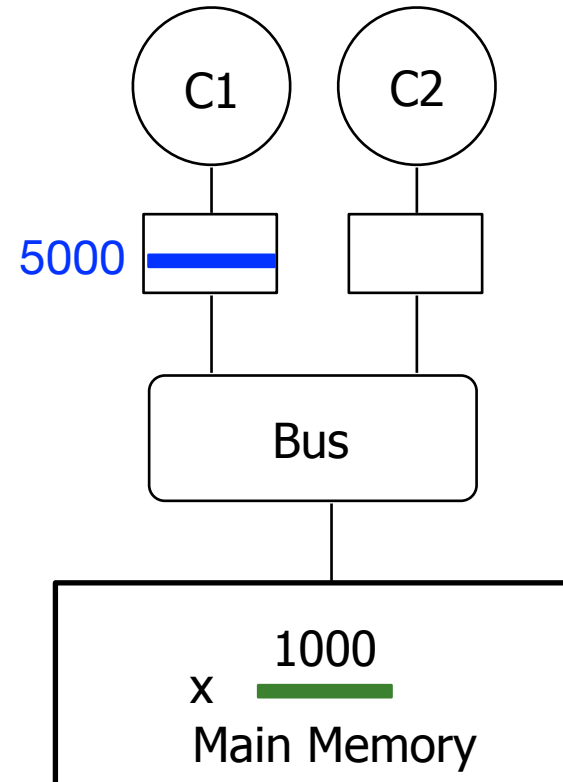
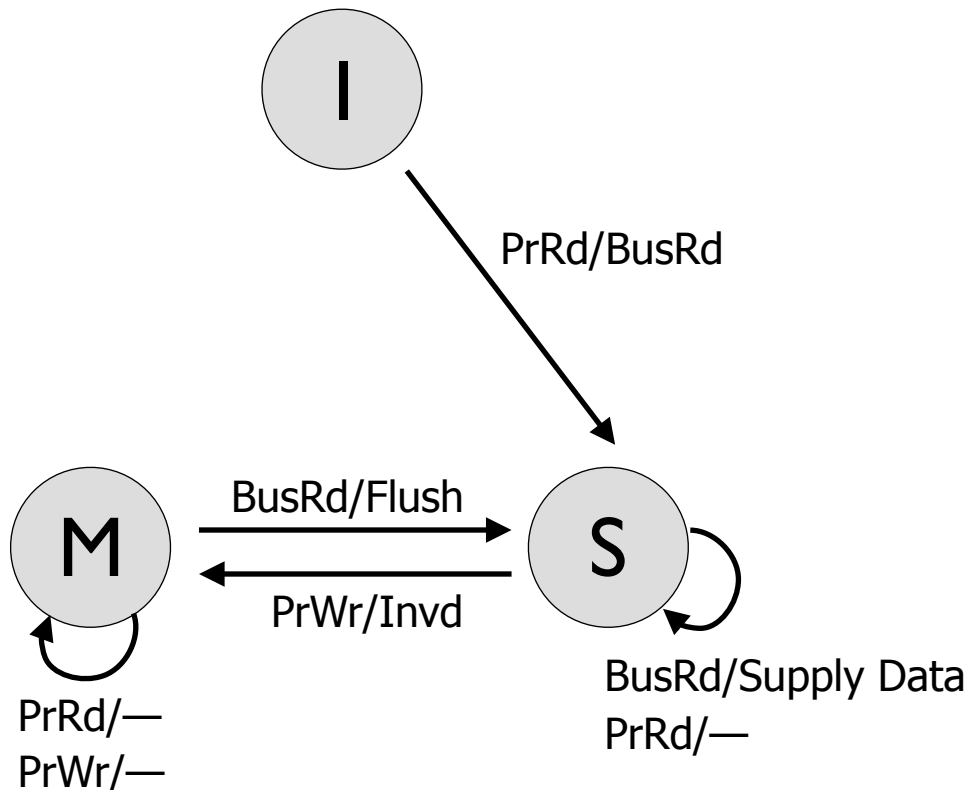


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Write:  $x = 7000$

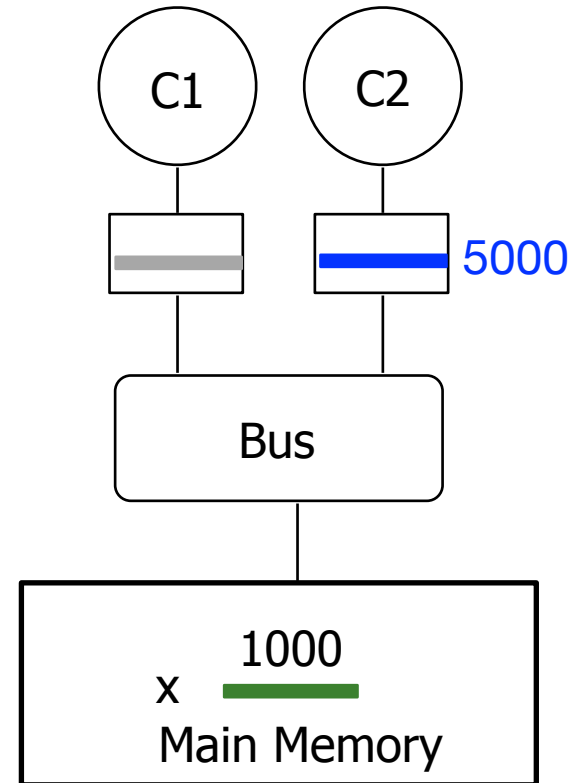
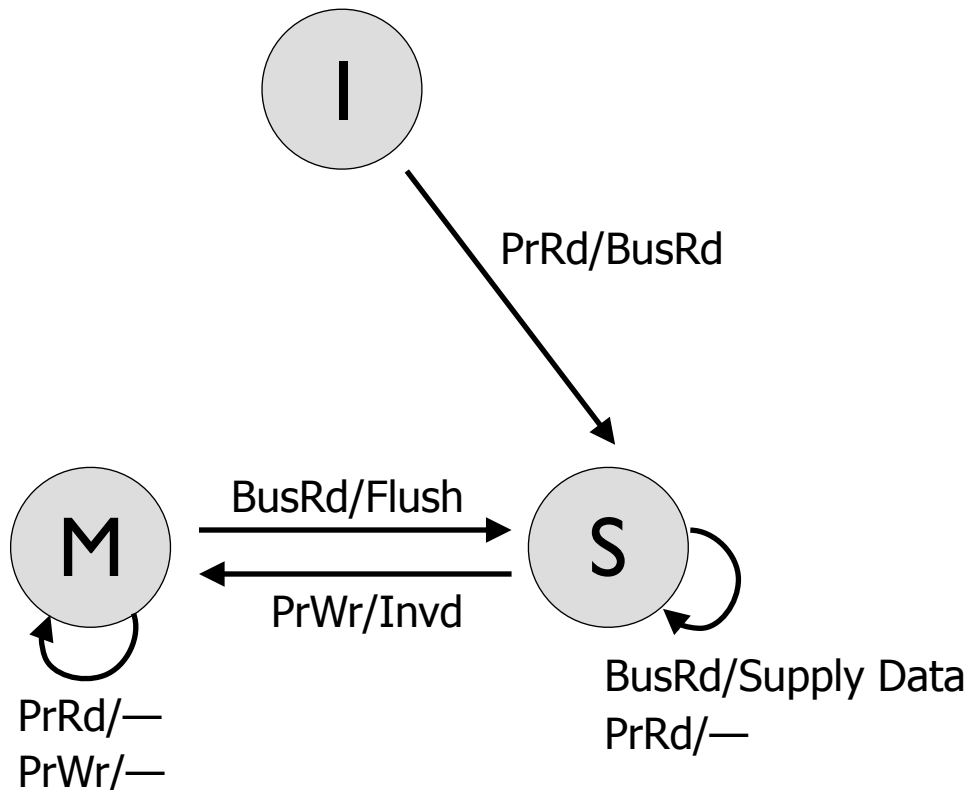


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Write:  $x = 7000$

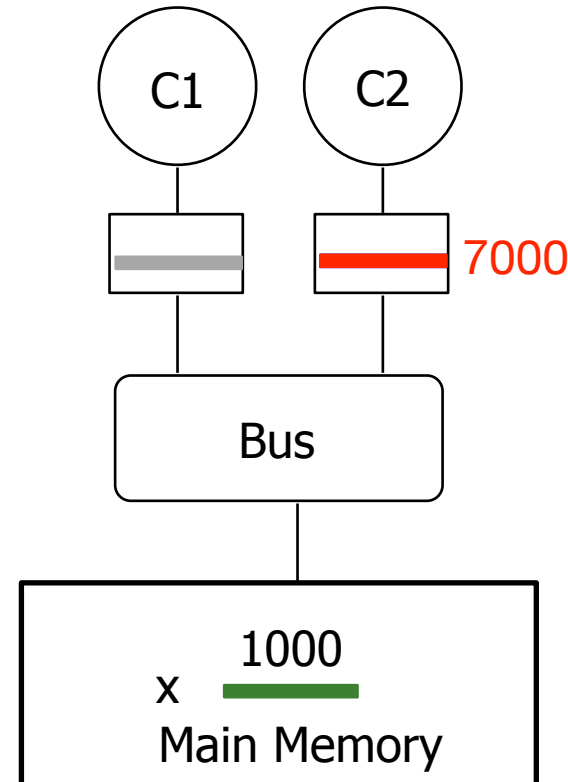
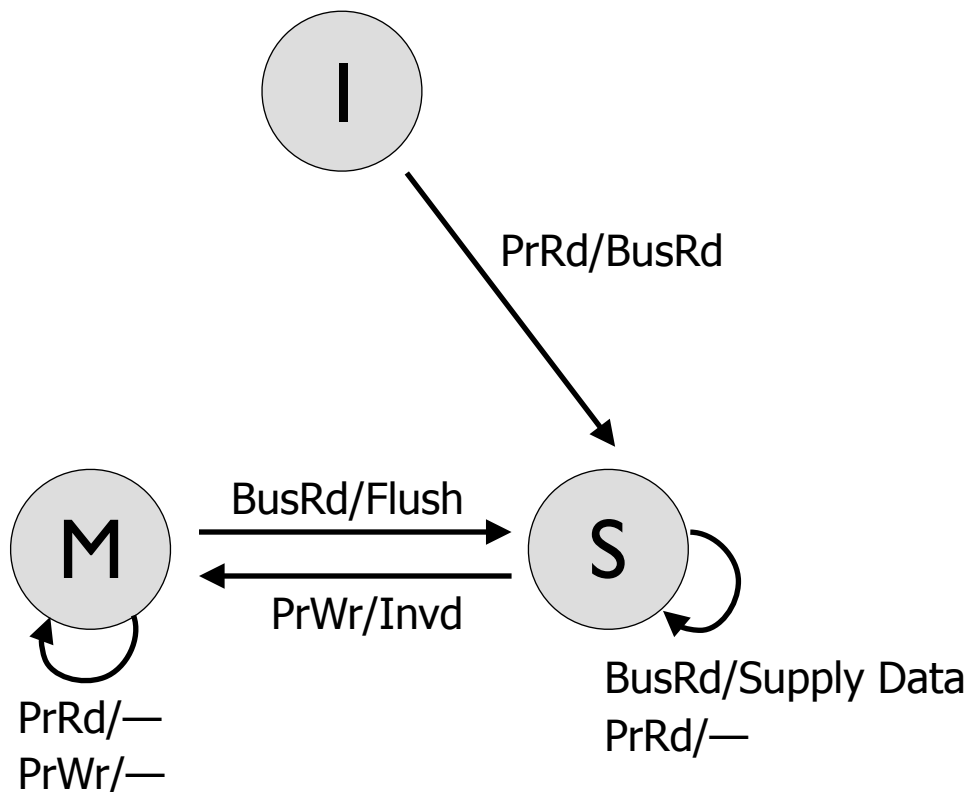


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Write:  $x = 7000$



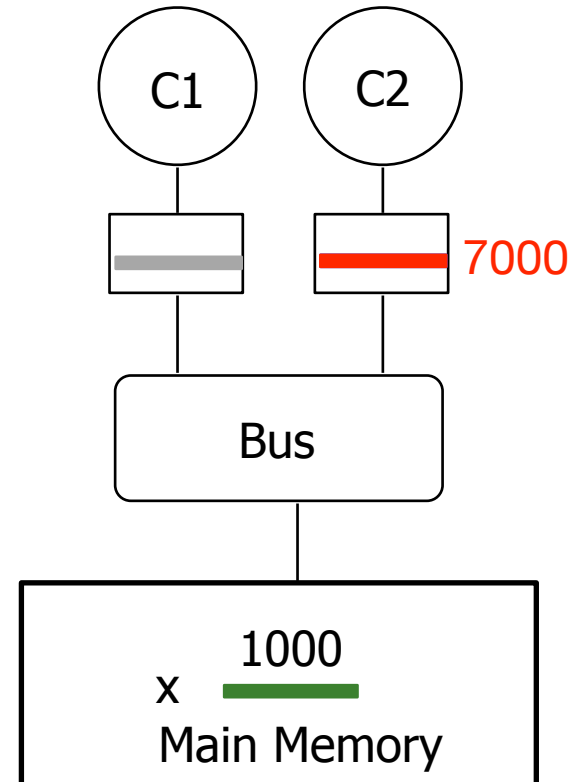
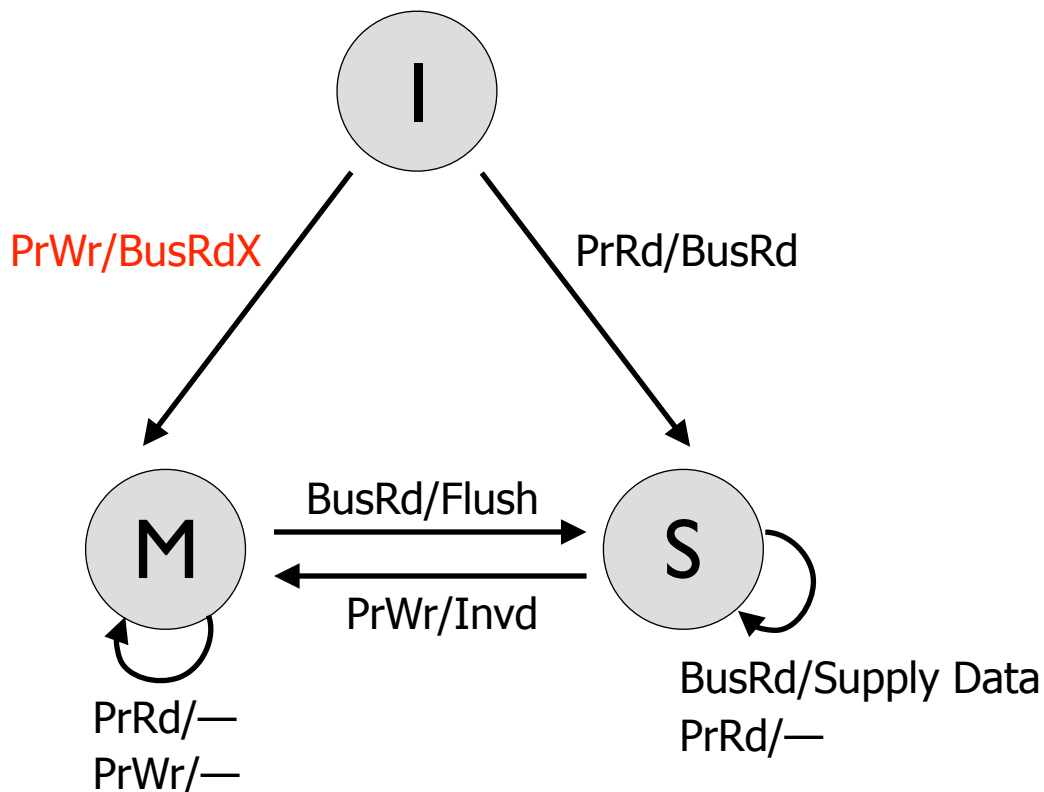


# Invalidate-Based Cache Coherence

Associate each cache line with 3  
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Write:  $x = 7000$

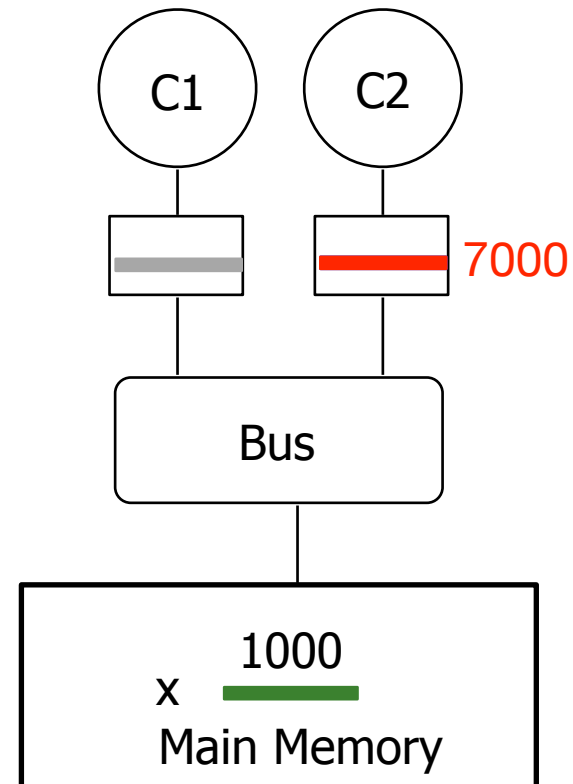
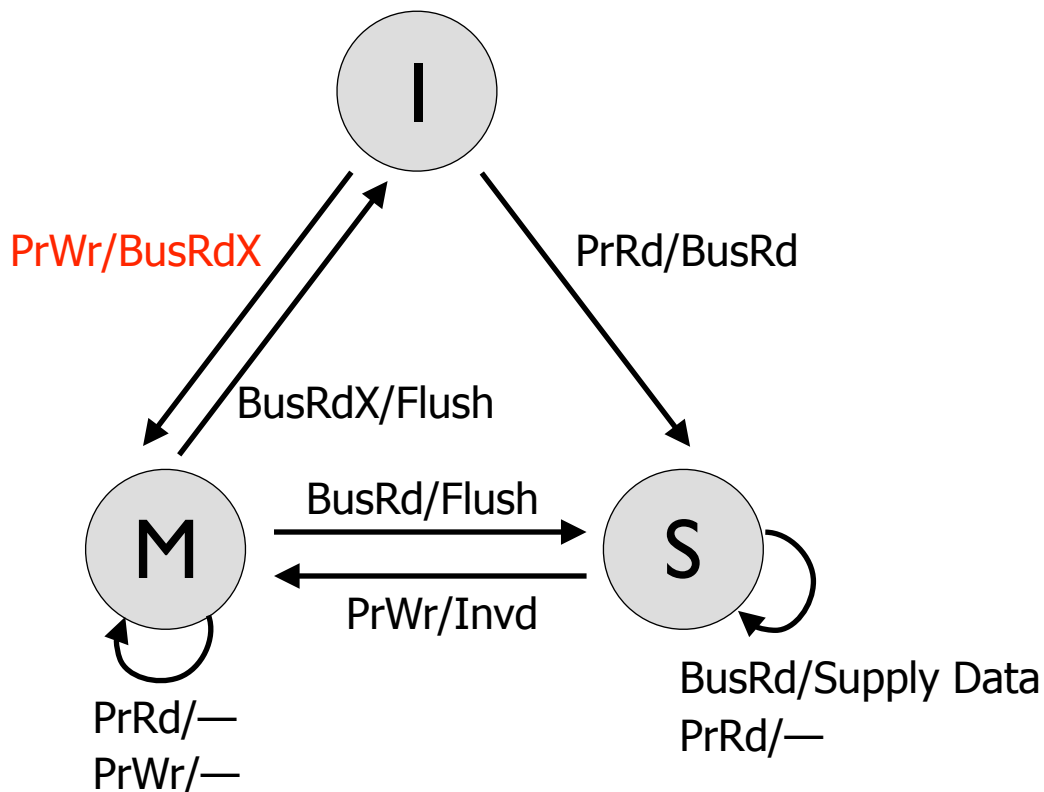


# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Write:  $x = 7000$

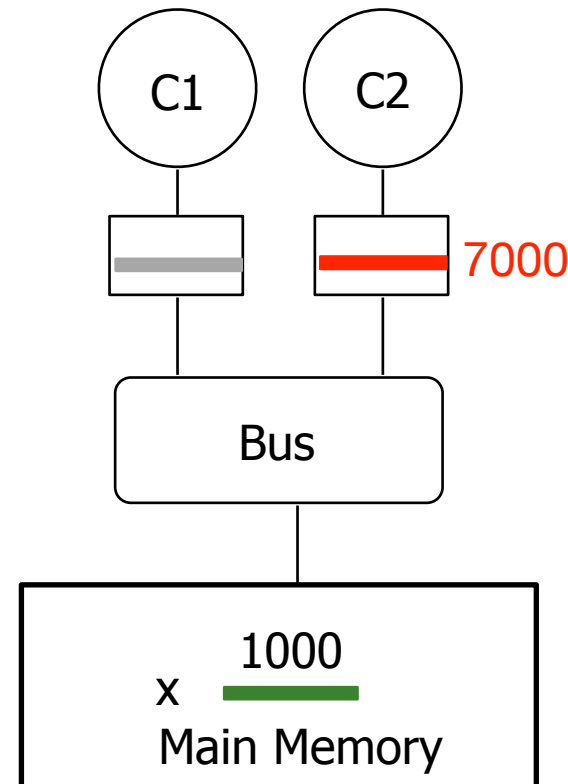
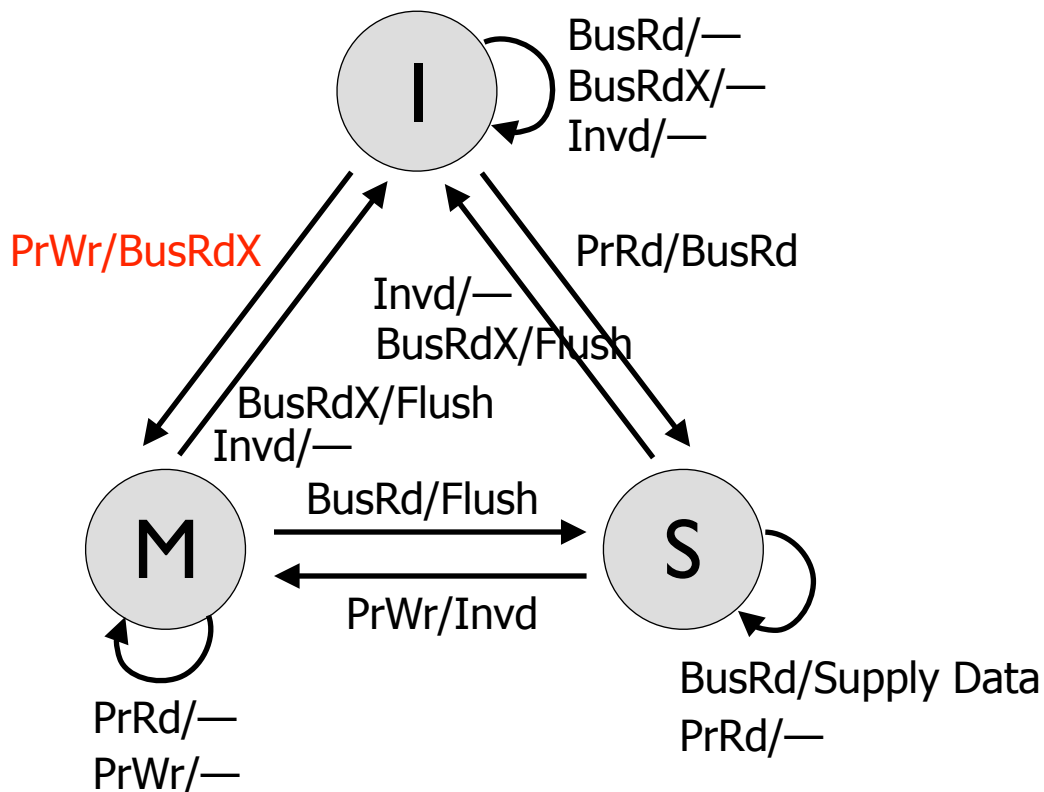


# Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for  $x$  in C2's cache;  
Syntax: Event/Action

Write:  $x = 7000$



# Readings: Cache Coherence

- Most helpful

- Culler and Singh, Parallel Computer Architecture
  - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
- Patterson&Hennessy, Computer Organization and Design
  - Chapter 5.8 (pp 534 – 538 in 4<sup>th</sup> and 4<sup>th</sup> revised eds.)
- Papamarcos and Patel, “[A low-overhead coherence solution for multiprocessors with private cache memories](#),” ISCA 1984.

- Also very useful

- Censier and Feautrier, “[A new solution to coherence problems in multicache systems](#),” IEEE Trans. Computers, 1978.
- Goodman, “[Using cache memory to reduce processor-memory traffic](#),” ISCA 1983.
- Laudon and Lenoski, “[The SGI Origin: a ccNUMA highly scalable server](#),” ISCA 1997.
- Martin et al, “[Token coherence: decoupling performance and correctness](#),” ISCA 2003.
- Baer and Wang, “[On the inclusion properties for multi-level cache hierarchies](#),” ISCA 1988.

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- Classic example: TLB
  - Hardware does not guarantee that TLBs of different core are coherent
  - ISA provides instructions for OS to flush PTEs
  - Called "TLB shutdown"



# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- Classic example: TLB
  - Hardware does not guarantee that TLBs of different core are coherent
  - ISA provides instructions for OS to flush PTEs
  - Called "TLB shutdown"

Take CSC 251/ECE 204 to learn more about advanced computer architecture concepts.