MultiProcessor and Thread-Level Parallelism

Chapter 6 - Computer Architecture : *A Quantitative Approach* - Hennessy & Patterson

Multi-Processor Architecture

- The **innovation** in uniprocessor architecture may **slow down** in near future.
- **Improved performance** may be obtained by using **more than one uniprocessors** in a system.
- The **progress in software technology** for **multiprocessor server and embedded applications** are **faster** due to **natural parallelism** available in the application.

Taxonomy

- **The design space of multiprocessor architecture is vast.**
- **Parallelism in instruction and data streams and Flynn's taxonomy - SISD, SIMD, MISD and MIMD.**

SISD

- **Single instruction stream and single data stream - uniprocessor system**

SIMD

- **Single instruction stream and multiple data stream.**
- **Each processing element has its own data memory but there is single instruction memory and a control processor.**
- **Vector processor architectures belongs to this class.**
- **The multimedia extension (MMX) and streaming SIMD extension (SSE) of Pentium are limited forms of SIMD parallelism.**

Limited SIMD: Pentium III

```
int main() {
 int i ;
 char d[] = {65, 66, 67, 68, 69, 70, 71, 72};
 char e[] = {32, 32, 32, 32, 32, 32, 32, 32};
 char f[8] ;

 for(i=0; i<8; ++i) printf("%c ", d[i]) ;
 printf("\n") ;
```

Limited SIMD: Pentium III

```
 asm(
     "movq     -16(%ebp), %mm1  #\n\t"
     "paddusb  -24(%ebp), %mm1  #\n\t"
     "movq     %mm1, -32(%ebp)  #\n\t"
     );
 for(i=0; i<8; ++i) printf("%c ", f[i]) ;
 printf("\n") ;
}
```

Limited SIMD: Pentium III

```
$ cc t4.c
$ a.out
A B C D E F G H
a b c d e f g h
$
```

MISD

- **Multiple instruction stream and single data stream.**
- **There is no commercial multiprocessor of this type.**

MIMD

- **Each processor has its own instruction stream and data stream.**
- **Each processor may be an off-the-shelf microprocessor.**

MIMD: The Architecture of Choice

- **MIMD is flexible and with the help of appropriate hardware and software, it can be used as a high performance computation platform for different applications.**
- **MIMD system can be manufactured using off-the-shelf components (microprocessors).**

Thread-Level Parallelism

- **Each processor executes independent processes, or communicating processes or different threads of a process.**
- **The threads or processes may be created by the programmer e.g. by fork() call or by the compiler e.g. parallel iterations of a loop.**

Two Classes of MIMD

- **Existing MIMD systems can be broadly classified into two classes.**
- **Centralized shared-memory architectures or symmetric shared-memory multiprocrssors (SMP) or uniform memory access (UMA) architectures.**
- **Distributed-memory multiprocessors.**

SMP

- **Small number of similar processors (at most a few dozen).**
- **Each processor has a large cache.**
- **A centralized memory (multiple banks) is shared through a memory bus.**
- **Each memory location has identical access time from each processor.**

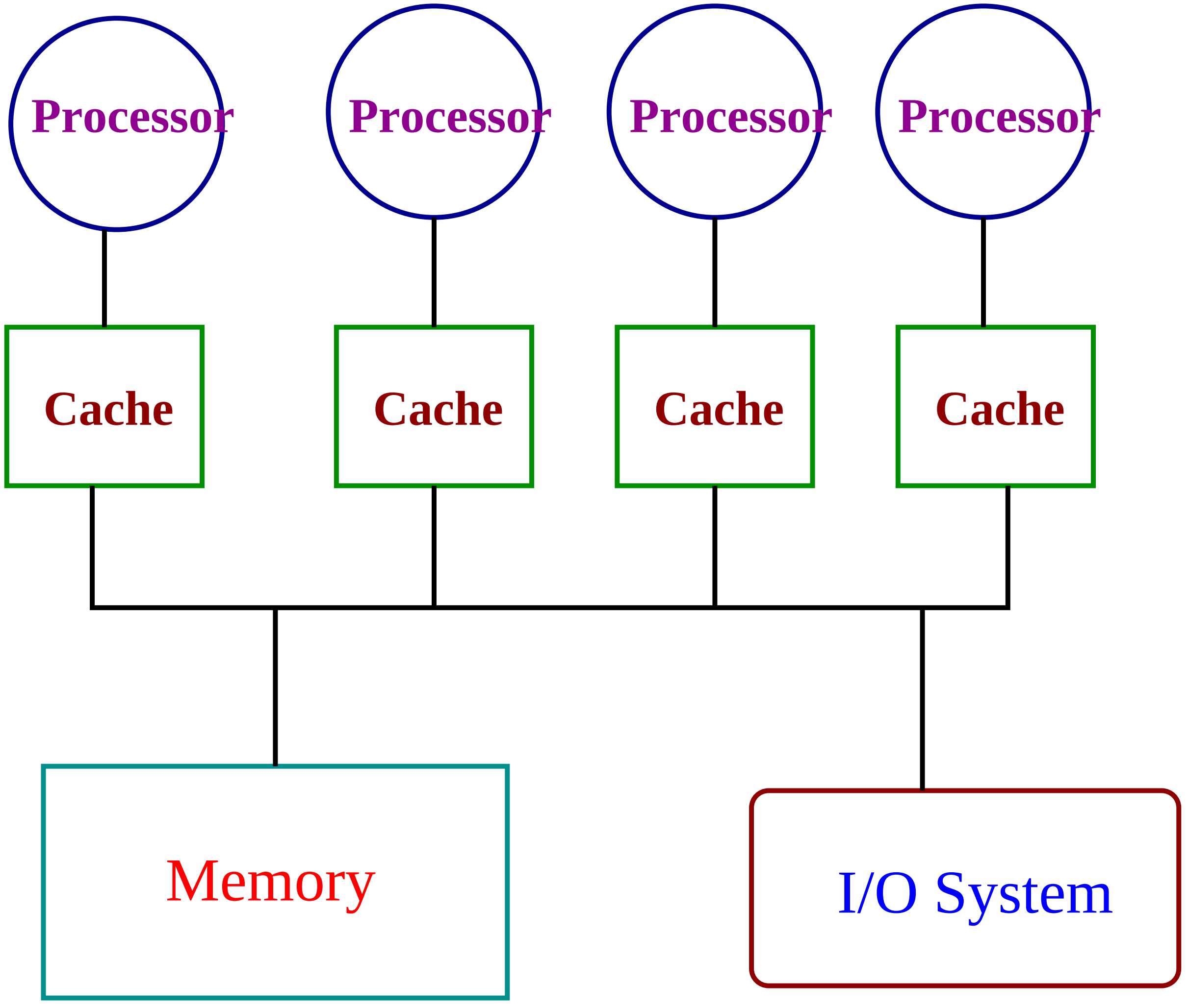
Processor
Processor
Processor
Processor
Cache
Cache
Cache
Cache
Memory
I/O System

Figure 1: **SMP**

Distributed Memory

- **Larger processor count.**
- **Memory is physically distributed among the processors for better bandwidth.**
- **Connected through high-speed interconnection e.g. switches.**

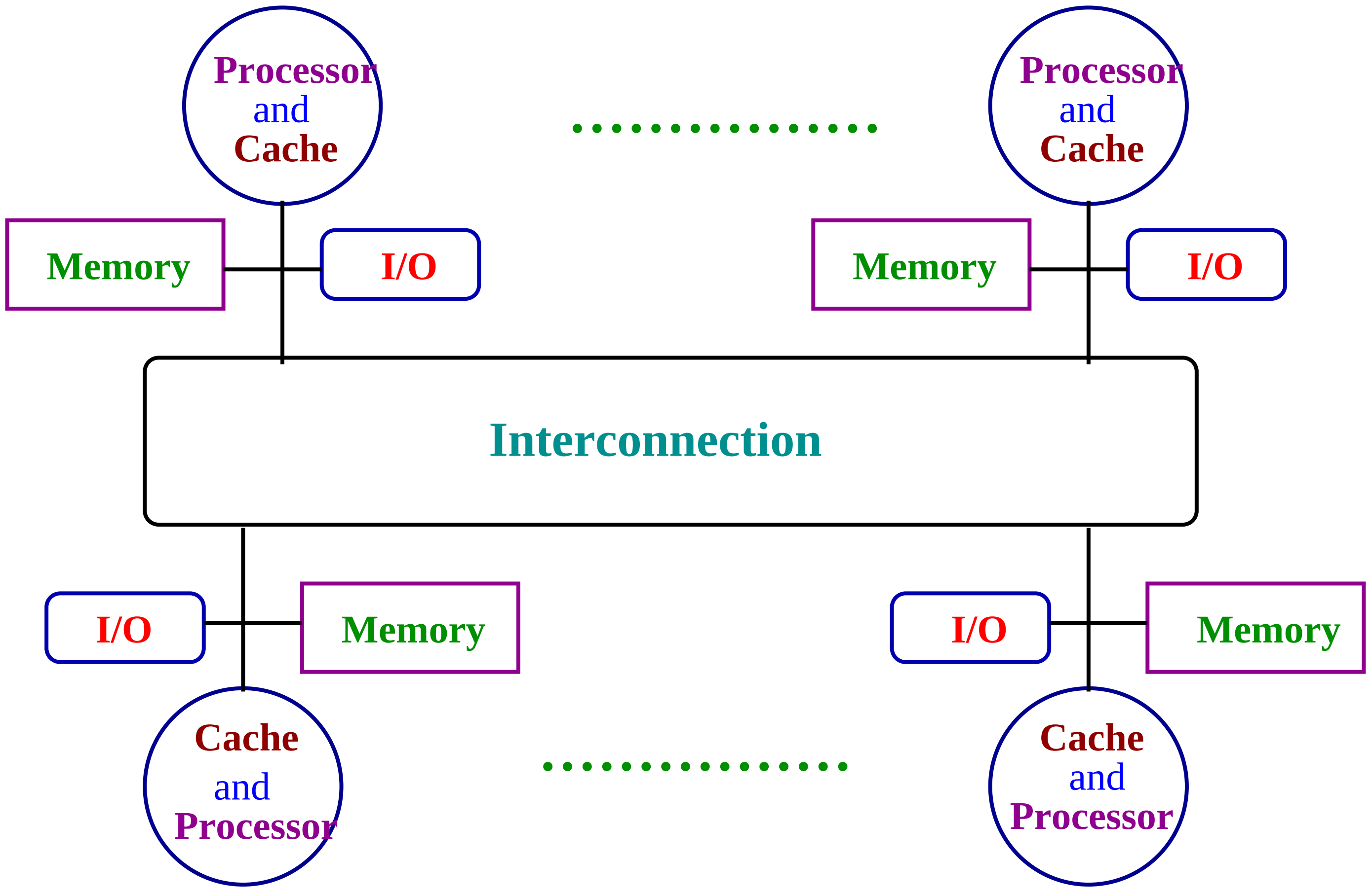

Figure 2: **Distributed Memory**

Distributed Memory

- The **bandwidth** for the **local memory is high** and the **latency is low**.
- But **access** to data present in the **local memory of some other processor** is **complex** and of **high latency**.

Memory Architecture

and

Models for Communication

- **Large-scale multiprocessors** have **physically distributed memory** with the processors.
- There are essentially **two different models of memory architectures** and the corresponding **models of communication**.

Memory Architecture

- **Memory is distributed with different processors** to support **higher bandwidth** demand of larger number of processors.
- **Any processor** can access a **location of physically distributed memory** (with proper access permission).
- This is called **distributed shared-memory architecture (DSM)** also known as **NUMA (nonuniform memory access)**.

Shared-Memory Architecture

- **The same physical address on two processors refers to the same location in memory.**
- **The communication is through the shared address space.**

Memory Architecture

- The other alternative is that **every processor has its private address space**.
- Each processor is essentially a **separate computer** - this is called a **multicomputer model** or **clusters**.
- The communication is by **message-passing**.
- Such cluster of processors use **standardized** or **customized interconnect** for communication

Message-Passing

- If a processor wants to **access (or process) some data** in a **remote memory**, it **sends a message** (similar to **remote procedure call (RPC)**).
- The **destination processor** receives it (**polling** or **interrupt**), performs the operation, and returns the result through a **reply message**.
- The message passing is **synchronous** - the initiating processor after sending the requests waits for the reply.

Message-Passing

- Communication may be **initiated** by the **producer of data**. The data produced can be sent to the consumer (no request).
- Such messages can be sent **asynchronously** and after sending, the produced may continue.
- A **consumer** may **block itself** if it tries to receive the message **before its arrival**.
- Similarly a **producer** may **block itself** if the message buffer is full.

Symmetric Shared-Memory Architecture

- Large and multilevel cache can reduce the main memory bandwidth.
- Multiple processors can share the same memory (may be multiple banks) if the bandwidth requirement of a single processor is reduced.
- The communication is through the read-write of shared data.
- Both shared and private data are cached.

Multiprocessor Cache Coherence

- Shared data copied to private cache of two different processors may give rise to cache coherence problem.

Time	Event	Cache_1	Cache_2	Memory
0				1
1	Read_{CPU_1}	1		1
2	Read_{CPU_2}	1	1	1
3	Write_{CPU_1}	0	1	0

Cache Coherence

- **A memory system is said to be coherent if a read from a memory location returns the last value written to the location.**
- **This definition is informal and simplified.**
- **Coherence: what value is returned by a read.**
- **Consistency: when a written value is returned by a read.**

Coherent Memory System

- A processor P writes a value v to a location L. If no other processor writes to the location L in between, then the processor P reads the same value v from L.
- If a processor P writes a value v to a location L, then a processor Q reads the value v from the location L, provided the read is sufficiently separated in time and there is no write in between.

Coherent Memory System

- **Write to the same location are serialized i.e. two writes to the same location are seen in the same order by every processor - if processor $P2$ writes to a location L after the processor $P1$ writes to the same location, every processor should see it in this order.**

Coherent Memory System

- These three conditions are **sufficient to ensure coherence**. But they do not say when the written value will be seen.
- If the **difference in time** between **a read after a write** in the same location is **very small**, it may be impossible to guarantee that the data read is the value written.

Memory Consistency Model

- The **memory consistency model** defines when a written value must be seen by a reader. This will be discussed afterward.
- We assume for simplicity that a write is not complete until all processors have seen the effect of it.
- Writes are performed in program order.

Enforcing Coherence

- A shared data may have copies on different caches of different processors - migration and replication of shared data is critical to performance (latency and bandwidth).
- The cache coherence is maintained by hardware protocols in small-scale multiprocessors - cache coherence protocols.
- There are essentially two types protocols to keep track of sharing status of data.

Cache Coherence Protocols

- **Directory based**: the **sharing status** of a block of data from the main memory is maintained globally in a **directory**.
- **Snooping**: each cache block maintains the **sharing status information** of the block. Each cache controller **monitors** or **snoops** the memory bus to see whether there is any request from other processor for a block present in its cache.

Snooping Protocol

- **Write Invalidate Protocol**: a **write** in a shared data block by a processor **invalidates** the copies of the block in all other cache.
- **Write Update/broadcast**: a **write** in a shared data block by a processor **updates** all the other cache copies of the data.
- The **write invalidate** is the **most common** protocol.

Write Invalidate Protocol

Processor	Bus	Cache_1	Cache_2	Memory
				0
Read_{CPU_1}	Miss	0		0
Read_{CPU_2}	Miss	0	0	0
Write_{CPU_1}	Invalidate	1		0
Read_{CPU_2}	Miss	1	1	1

Write Invalidate Protocol

- A **write by a processor** ($P1$) **invalidates** the corresponding cache contents of **other processors** ($P2$).
- The block is written back when there is a read/write request from another processor (**write-back** and **write-allocate** cache).
- Only one processor gets the exclusive write access to a block. If two processors try to write simultaneously, one of them wins. The second processor should get the updated copy of the data before it writes - **write serialization**.

Write Update/Broadcast Protocol

Processor	Bus	Cache_1	Cache_2	Memory
				0
Read_{CPU_1}	Miss	0		0
Read_{CPU_2}	Miss	0	0	0
Write_{CPU_1}	Broadcast	1	1	1
Read_{CPU_2}	Hit	1	1	1

Write Invalidate vs Write Update

- **Multiple writes** in the same cache block by the same processor require **multiple broadcasts** for updates, but a **single invalidation** on the **first write**.
- The **delay** between **writing** a word by a processor and then **reading** the word by another processor is usually **less in write update** as the new data is **immediately updated** in processors having its copy.
- Invalidation protocol generates **less traffic in the memory bus**.

Write Invalidate: Implementation

- In small-scale multiprocessor, the bus is used to invalidate - the address to invalidate is broadcasted.
- All processors snoop on the bus watching addresses. If the address match with its cache block, it invalidates the block.

Write Invalidate: Implementation

- The **write serialization** is achieved through **serialization of bus access**. The first processor that gets access to the bus, writes in the cache and **other copies of data are invalidated**.
- Write to a shared data is **not complete** unless the bus access is obtained.

Write Invalidate: Implementation

- The snooping scheme can also be used for cache miss.
- If there is a read request on the bus and a cache controller detects that the requested dirty cache block is available in its cache; it sends it to the bus. The data not only goes to the requested cache but may also be written-back in the main memory.

Write Invalidate: Implementation

- The normal tag may be used for snooping.
- The valid bit is used to invalidate a block.
- Write for a private data in a write-back cache need not be placed on the memory bus - this saves write time and bus bandwidth.

Write Invalidate: Implementation

- An **extra state bit per block** may be used for **sharing.**
- **Write in a shared block generates an invalidation** on the bus and **marks the block private** (**owner**). Another write in the same block by the same processor does not generate invalidation.
- If **another processor** requests for this block, the **owner** sends the data on the bus (**write-back**) and changes the state to **shared**.

Write Invalidate: Implementation

- **The cache-tage is compared on every memory bus transaction and that may interfere with the normal CPU access of tag.**
- **The tag may be duplicated or multilevel cache may be used.**

Write Invalidate: Implementation

- **A bus-based cache-coherence protocol is implemented by an FSM for every processor.**
- **The FSM gets requests from the processor and also from the bus.**

Write Invalidate: An Example

We assume the following facts in the example.

- **We do not distinguish between a write hit and a write miss in a shared cache block and treat them as write miss.**
- **When a write miss is placed on the bus, any processor with a copy of the block invalidates it.**
- **In a write-back cache, if the block is exclusive and there is a write miss, the existing block is written back.**

Write Invalidate: An Example

Request	Source	State of Addr. Cache Block	Expl
Read_{Hit}	CPU	Shared/Excl.	Read data from cache
Read_{Miss}	CPU	Invalid	Read miss on bus
Read_{Miss}	CPU	Shared	Read miss on bus
Read_{Miss}	CPU	Excl.	Write-back, then read miss on bus

Write Invalidate: An Example

Request	Source	State of Addr. Cache Block	Expl
Write_{Hit}	CPU	Excl.	Write data in cache
Write_{Hit}	CPU	Shared	Place write miss on bus, write data in cache
Write_{Miss}	CPU	Invalid	Place write miss on bus, fetch the cache block, write data

Write Invalidate: An Example

Request	Source	State of Addr. Cache Block	Expl
Write_{Miss}	CPU	Shared	Place write miss on bus, fetch the cache block, write data
Write_{Miss}	CPU	Excl.	Write back, place write miss on bus, fetch the cache block, write data

Write Invalidate: An Example

Request	Source	State of Addr. Cache Block	Expl
Read_{Miss}	bus	Shared	No action
Read_{Miss}	bus	Excl.	Place data on bus, change block to shared
Write_{Miss}	bus	Shared	Invalidate the block
Write_{Miss}	bus	Excl.	Write back, invalidate the block

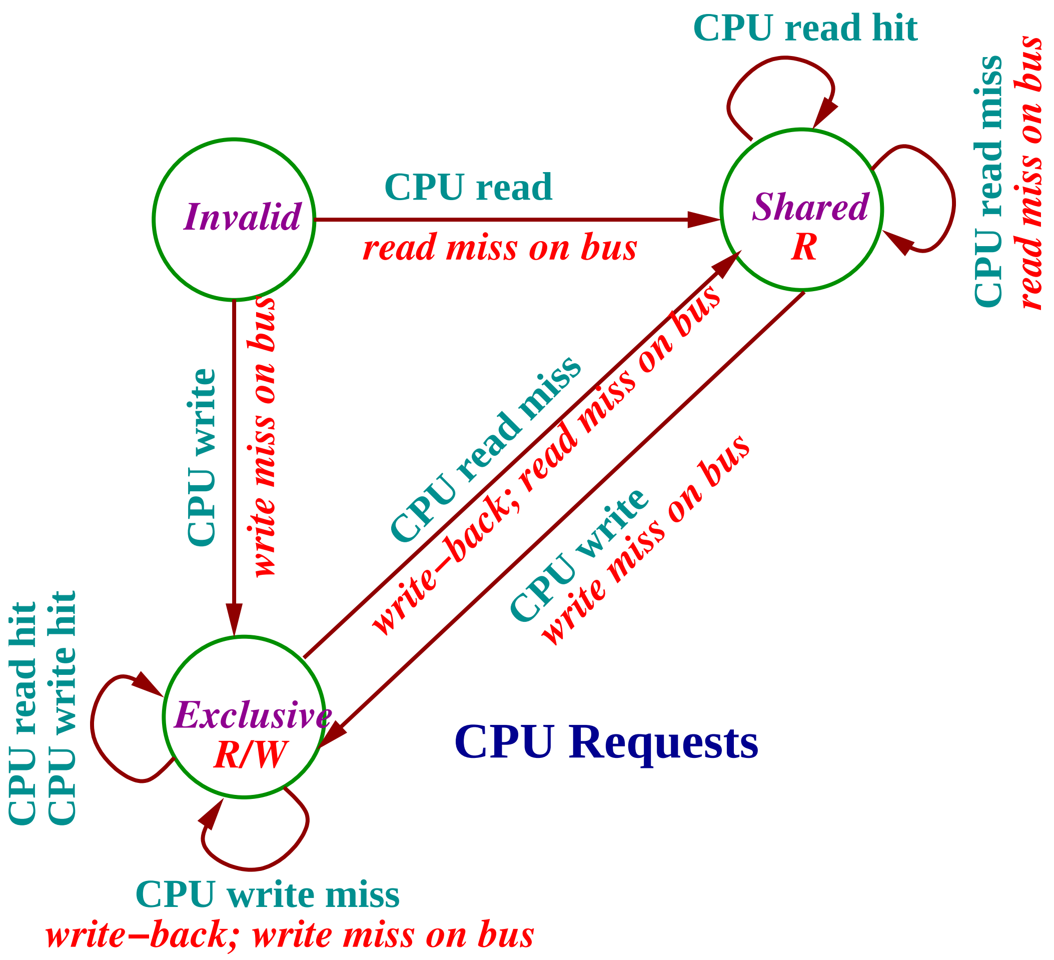

Figure 3: **Cache Coherence FSM**

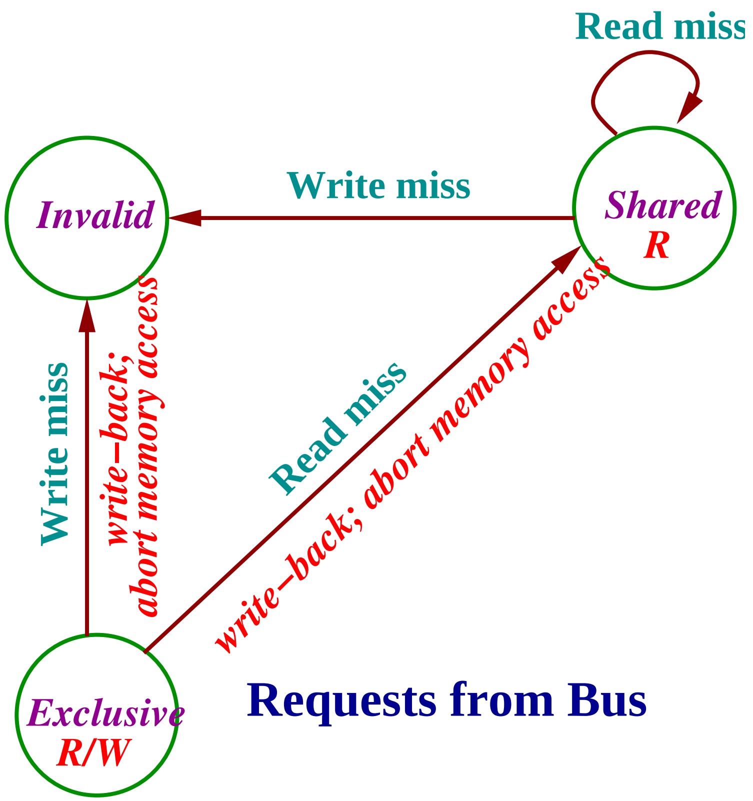

Figure 4: **Cache Coherence FSM**

CPU Requests

- Cache block is in exclusive state:
  - There is no state change on CPU **read** or **write hit**.
  - If there is a CPU write miss, the block contains **exclusive data** for some **other address**. The block is written-back and a write miss for the address is placed on the bus which **invalidates** the **same block** present in other cache. **Write of the block is completed.**

CPU Requests

- Cache block is in **exclusive** state:
  - If there is a **CPU read miss**, the block contains **exclusive data** for some **other address** which is to be **written-back**. After that the **read miss for the address** is placed on the bus. The state changes to **shared** as the data may be present in another cache.

CPU Requests

- Cache block is in **shared** state:
  - **No state change** on **read hit** or **read miss**.
  - In case of **read miss**, a read miss is placed on the bus (no **write-back** as the data is shared).
  - In case of **write hit** or **miss** (both treated as a miss in a shared state) **write miss** is asserted on the bus, state is changed to **exclusive** and write is completed.

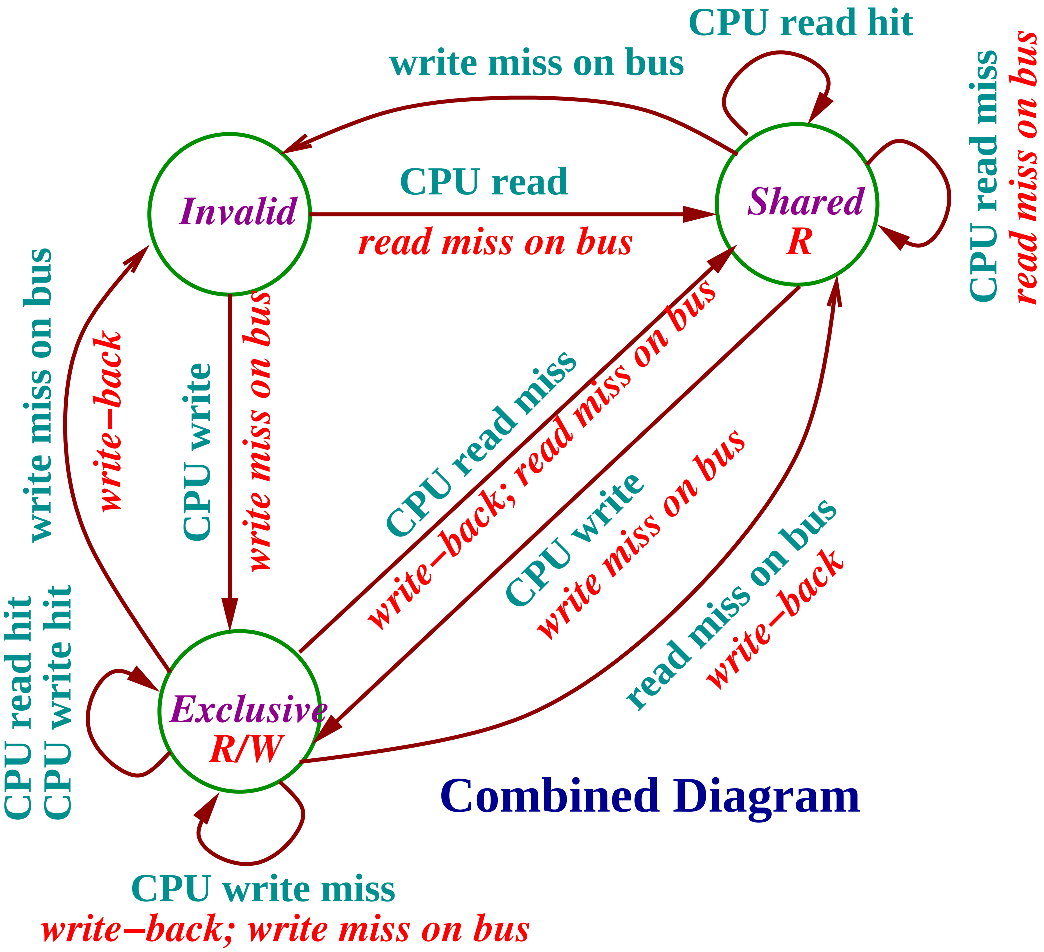

Figure 5: **Cache Coherence FSM**

Snooping Protocol

- The protocol is correct but simplified. Actual implementation is more tricky.
- Operations like write miss is not atomic - in case of a write miss the FSM does the following:
  - Detects the write miss,
  - Gets the bus and asserts the write miss,
  - Gets the most recent value, and
  - Writes the data in the cache block.

All these cannot be completed in one clock.

Write Miss is not Atomic

- **Write miss** is decomposed into steps (separated in time).
- The first step - detects the miss and requests the bus.
- The second step - acquires the bus, places the miss on the bus, gets the block and writes the data.
- The cache block is not exclusive before the second step starts.

Write is Serialize

- The bus transaction is atomic (not a split transaction).
- The **write to a cache block is serialized** if the **block is not made exclusive before the second step** as mentioned and block is not written before acquiring the bus.

New States

- **New states are all transient states - the controller waits in these states to acquire the bus.**
- **Four (4) states are for pending write backs.**
- **One (1) is for pending read and the other (1) is for pending write.**

Write Backs

- A **write miss** on the bus by another processor for an **exclusive block** here.
- A **read miss** on the bus by another processor for an **exclusive block** here.
- A **CPU read miss** on an **exclusive block**.
- A **CPU write miss** on an **exclusive block**.

Replicating Controller

- The FSM can be **logically replicated** for **differnt cache blocks** if
  - an operation on the bus for a cache block and a pending operation for a differnt cache block are noninterfering.
  - The controller correctly deals with the case when a pending operation and a bus operation are for the same block.