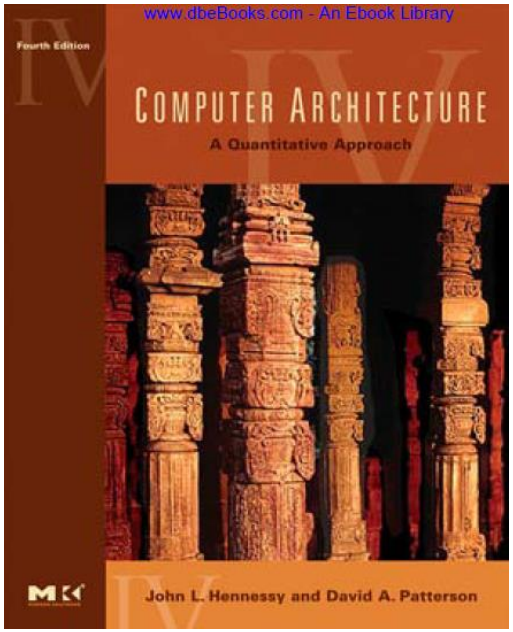
Multi Processing

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University



Introduction

Since mid 2000s uniprocessor performance increase slows down:

- diminishing returns in exploiting ILP
- growing concern over power
- diminishing performance improvement of transistors

Since 1990s designers sought a way to build servers and supercomputers achieving higher performance than a single microprocessor.

A new era in computer architecture, where multiprocessors play a major role.

Taxonomy

Flynn [1966] proposed four categories:

1. **Single Instruction stream, Single Data stream (SISD)**—Ordinary uniprocessor.

2. **Single Instruction stream, Multiple Data streams (SIMD)**—Same instruction executed by multiple processors using different data streams.

Each processor has its own data memory, but there is a single instruction memory and control processor, which fetches and dispatches instructions.

Popular in multimedia, graphics, vector operations.

3. **Multiple Instruction streams, Single Data stream** (**MISD**)—Not practical, no commercial.

4. **Multiple Instruction streams, Multiple Data streams** (**MIMD**)—Each processor fetches its own instructions and operates on its own data.

Exploits **thread-level parallelism** , since multiple threads operate in parallel.

Thread-level parallelism is more flexible than data-level parallelism and thus more generally applicable.

MIMD is the most popular. The increasing silicon capacity enables placing multiple processors on a single die, called **multicore**.

Multicore share L2, L3 cache or memory and I/O buses.

2000's processors, as IBM Power5, Sun T1, and Intel Pentium D and Xeon-MP, are multicore.

Multicore which relies on **replication** is advantageous over wide superscalar.

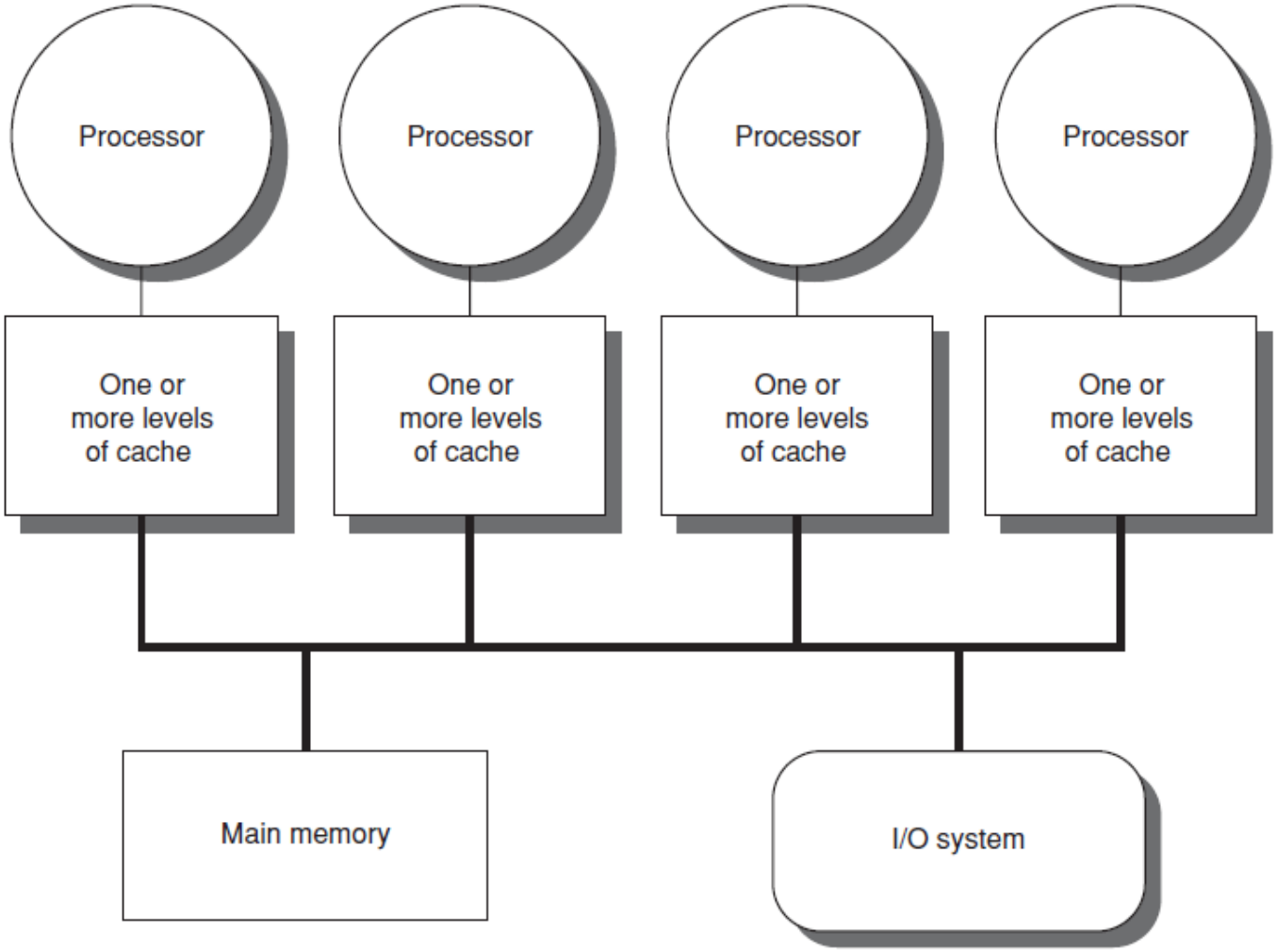
Each processor is executing its own instruction stream.

To take advantage of a MIMD multiprocessor with n processors, we should have at least n threads or processes to execute.

The independent threads within a single process are typically identified by the programmer or created by the compiler.

The threads may come from large-scale, independent processes scheduled and manipulated by OS.

Centralized Shared-Memory



With large caches, a single memory can satisfy the memory demands of a small number of processors.

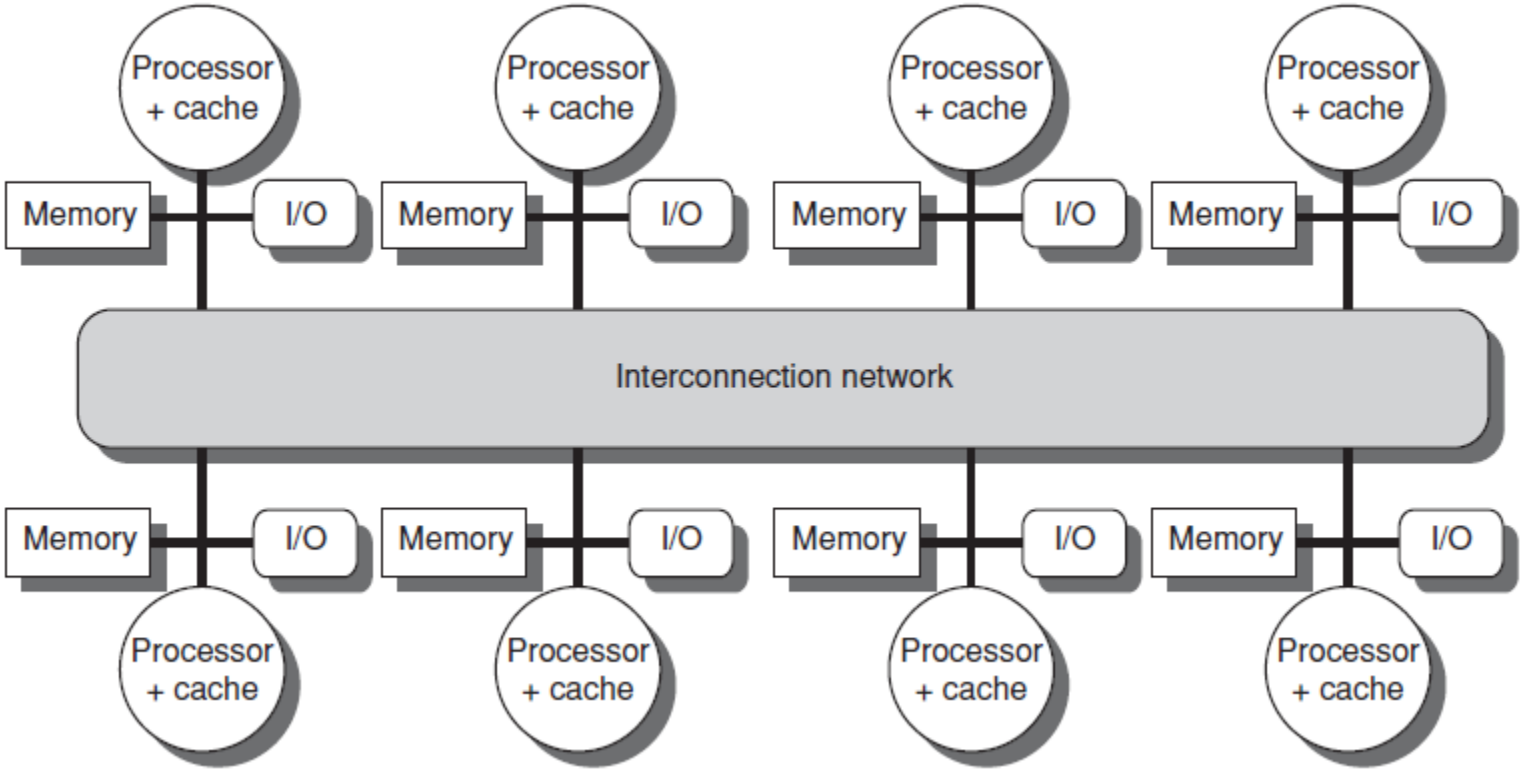
By using appropriate network it can be scaled to a few dozen processors.

Because the main memory is symmetric to all processors, these are called **Symmetric** (shared-memory) **Multiprocessors** (**SMPs**).

Due to the uniform access time from any processor, it is sometimes called **Uniform Memory Access** (**UMA**).

Another group consists of multiprocessors with physically **distributed memory**.

Distributed Memory



To support many processors, memory is distributed among the processors rather than centralized.

Otherwise the memory system would not be able to support the bandwidth demands without incurring excessively long access latency.

The larger number of processors also raises the need for a high-bandwidth interconnect.

Distributed memory is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node.

A disadvantage is that communicating data between processors becomes complex, requiring more effort in the SW.

Communication in Distributed Memory

In **Distributed Shared-Memory** (**DSM**) architectures, communication occurs through a shared address space, as in a symmetric shared-memory.

The **physically separate** memories are addressed as **one logical** address space, so memory reference can be made by any processor to any memory location.

Alternatively, the address space can consist of multiple, **logically disjoint**, private address spaces, which cannot be addressed by a remote processor.

The same physical address in different processors refers to different locations in different memories.

For a multiprocessor with a **shared address space**, the address space is used to communicate data implicitly via load and store operations.

Multiprocessor with multiple address spaces communicates data by explicitly passing messages among the processors, called **message-passing multiprocessors**.

Amdahl's Law makes parallel processing challenging. A hurdle is the **limited parallelism** in programs.

Example. We want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

The program operates in two modes:

- parallel with all processors fully used (enhanced) or
- serial with only one processor in use.

Answer. Amdahl's Law states

$$\text{Speedup} = \frac{1}{\frac{\text{Frac parallel}}{\text{Speedup parallel}} + (1 - \text{Frac parallel})}$$

$$0.8 \times \text{Frac parallel} + 80 \times (1 - \text{Frac parallel}) = 1$$

$$\text{Frac parallel} = 0.9975!$$

To achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential. ■

To achieve linear speedup, the entire program must be parallel with no serial portions. Not realistic.

A second hurdle arises from the relatively high cost of communications, from 50 to over 1000 clock cycles.

It depends on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor.

Symmetric shared-memory machines support the caching of both **shared** and **private** data.

Private data are used by a single processor.

shared data are used by multiple processors, providing communication among the processors through reads and writes of the shared data.

When a **private** item is cached, since no other processor uses the data, the program behavior is identical to uniprocessor.

When **shared** data are cached, the shared value may be replicated in multiple caches.

Advantages:

- Reduction in **access latency**
- Reduction in required **memory bandwidth**
- Reduction in **contention** that exists for shared data items read by multiple processors simultaneously.

New problem: **cache coherence**.

The view of memory held by two different processors is through their individual caches, which could end up seeing two different values.

Cache Coherence

A memory system is **coherent** if:

1. **Preserves program order**. A read by P_1 to X after a write by P_1 to X, with no intervening writes of X by P_2, always returns the value written by P_1.

It is expect also in uniprocessors.

2. A **read** by P_1 to X **after** a **write** by P_2 to X returns the value written by P_2 if the read and write are **sufficiently separated** in time and no other writes to X interven.

If a write of X on P_1 and read of X on P_2 are **insufficiently separated** in time, it may be **impossible** to ensure that P_2 read returns X written by P_1, since X may not have left P_1 yet.

3. **Serialization.** Two writes to the same location by any two processors are seen in the same order by all processors.

The issue of exactly **when** a written value must be seen by a reader is defined by **memory consistency model**.

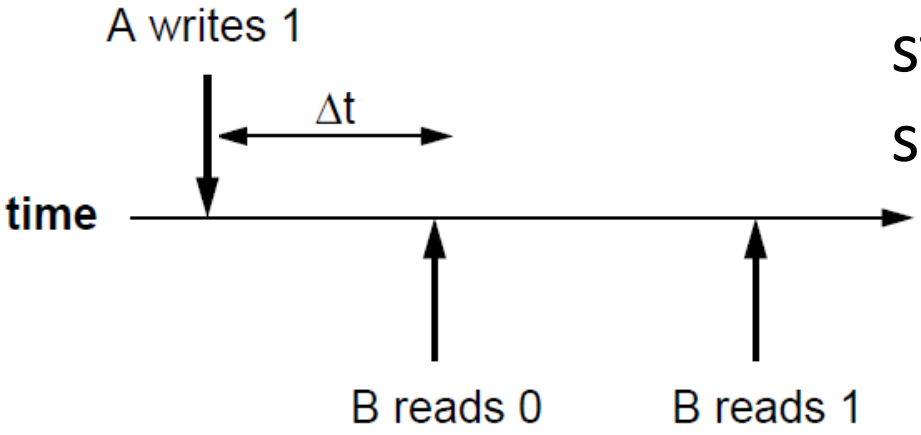
We assume that a **write does not complete** (allow next write) until all processors see the effect of that write.

We assume that the processor does not change the order of any write with respect to any other memory access (different memory locations).

These two conditions mean that if a processor writes location A followed by location B, any processor seeing the new value of B must also see the new value of A.

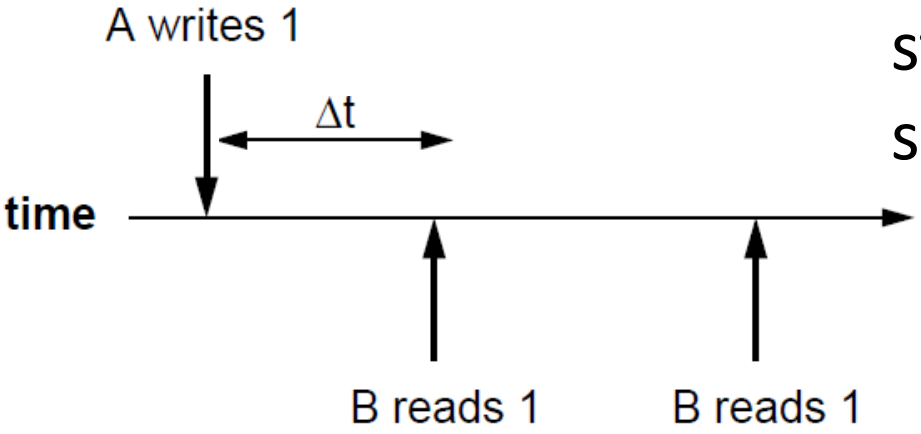
These restrictions allow the processor to **reorder reads**, but forces it to finish a **write** in **program order**.

Memory Consistency



strict consistency **x**

sequential consistency **v**



strict consistency **v**

sequential consistency **v**

Consistency Implications

```
P1:  (initially, A=0)
     A=1;
     if (B==0) {
         kill P2;
     }
```

```
P2:  (initially, B=0)
     B=1;
     if (A==0) {
        kill P1;
     }
```

Sequential consistency allows 0 or 1 processes to be killed, but not both.

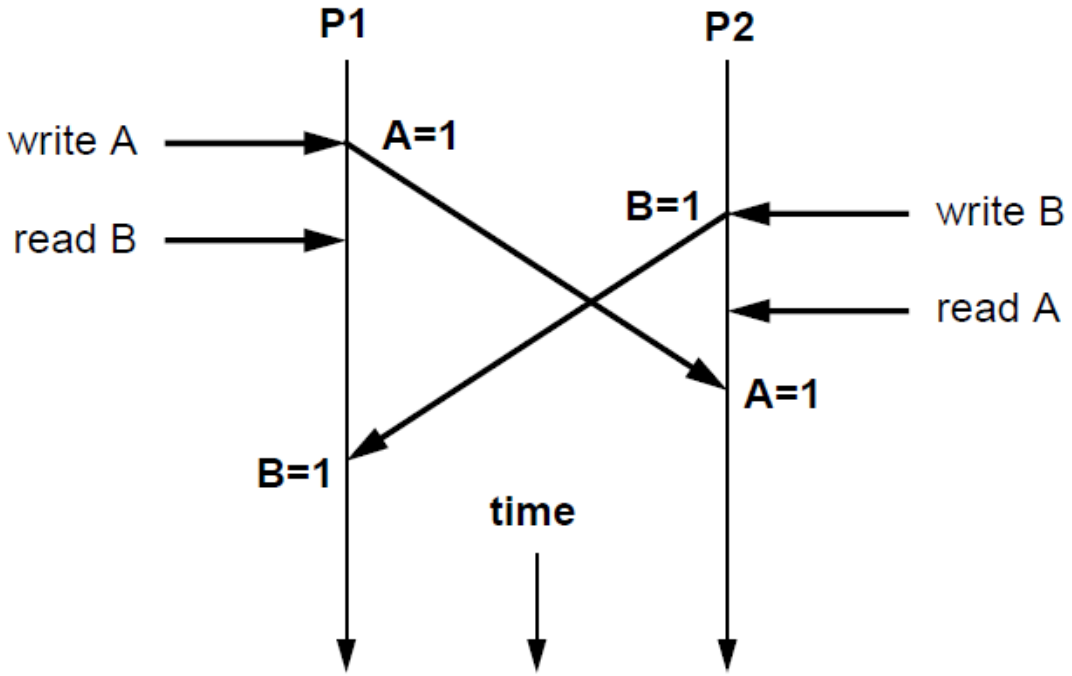
P1 will only try to kill P2 if P1's read to B occurs before P2's write to B.

The in-order execution of memory events implies that P1's write to A must come before P2's read of A.

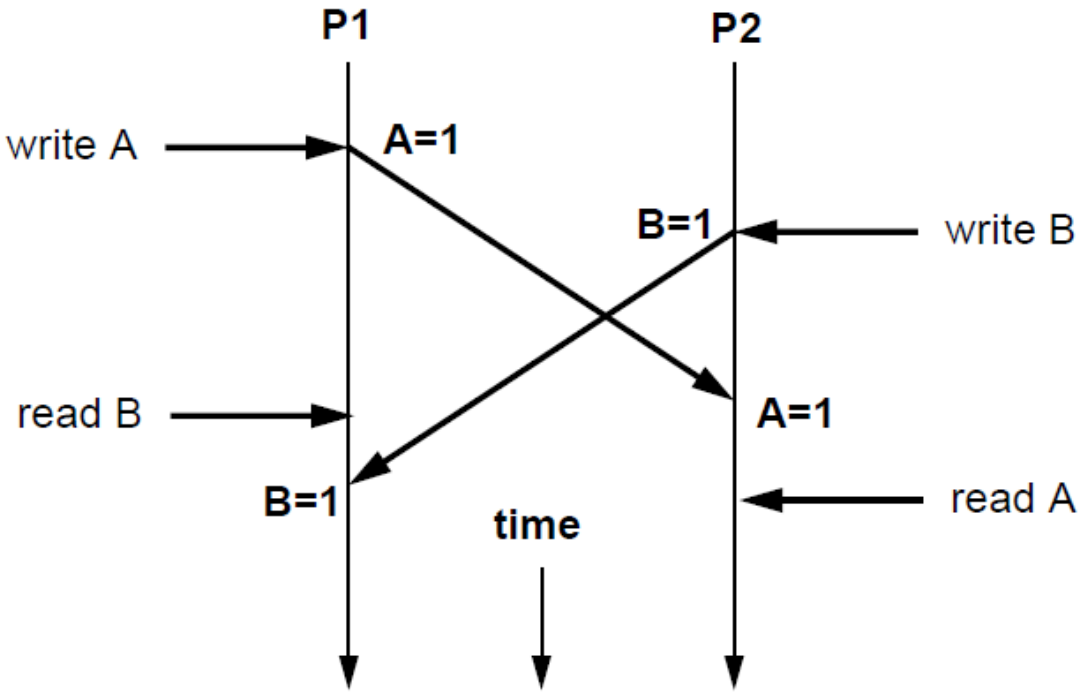
P1 and P2 try to kill each other, disallowed by sequential model.

```
P1:  (initially, A=0)
     A=1;
     if (B==0) {
        kill P2;
     }
```

```
P2:  (initially, B=0)
     B=1;
     if (A==0) {
        kill P1;
     }
```



Sequential consistency must delay all memory operations following a write until the write is observed by all other clients.



Can also be solved by speculation.

Execute memory instructions subsequent to write, but hold commitment long enough after write to ensure that all processor cores observe the write.

If the write observed conflicts with an early read, that instruction's commitment must be halted, its results must be discarded, and the instruction must be re executed in light of the new data.

Coherence Enforcement

A program running on multiple processors will normally have copies of the same data in several caches.

The protocols to maintain coherence for multiple processors are called **cache coherence protocols**.

Cache coherence protocol implementation must **track** the state of **any** sharing of a **data block**.

Two classes of protocols. A **directory based** keeps the sharing status a block of physical memory in one location, called the **directory** (not discussed).

Snooping protocols have no centralized state.

Every cache having a copy of a block of physical memory has a copy of the block's sharing status (status bits).

All caches are accessible via a bus or switch, and all cache controllers **snoop** (monitor) to determine whether or not they have a copy of a block that is requested for access.

Write invalidate protocol is a method ensuring that a processor has **exclusive access** to a data item before it writes that item, invalidating other copies on a write.

Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs. All **other** cached **copies** of the item are **invalidated**.

Consider a write followed by a read by another processor.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

If two processors do attempt to write the same data simultaneously, one of them wins the race, invalidating the other processor's copy.

To complete its write the other processor must obtain a new copy of the data, now containing the new value. This protocol enforces **write serialization**.

The **alternative** to an **invalidate** protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a **write update** or **write broadcast** protocol.

Write Invalidate Implementation

For invalidation, the writing processor **acquires bus** access and **broadcasts** the **address** to be invalidated. All processors continuously snoop on the bus, watching the addresses.

The processors check whether the **address** on the bus is in **their cache**. If so, the corresponding data in the cache are **invalidated**.

If two processors attempt to write shared blocks at the **same time**, their attempts to broadcast invalidation are serialized when they arbitrate for the bus.

We also need to **locate** a data item when a **cache miss** occurs. In a **write-through** cache, it is easy to find the recent value in the memory.

The problem of finding the most recent data value in **write-back cache** is harder, since most recent data item is in a cache. Write-back caches use **snooping** both for cache **misses** and for **writes**.

Each processor snoops every address placed on the bus. If it finds to have a **dirty copy** of the requested block, it provides that block in response to the read request and **aborts** the memory access.

Snooping Protocol Example

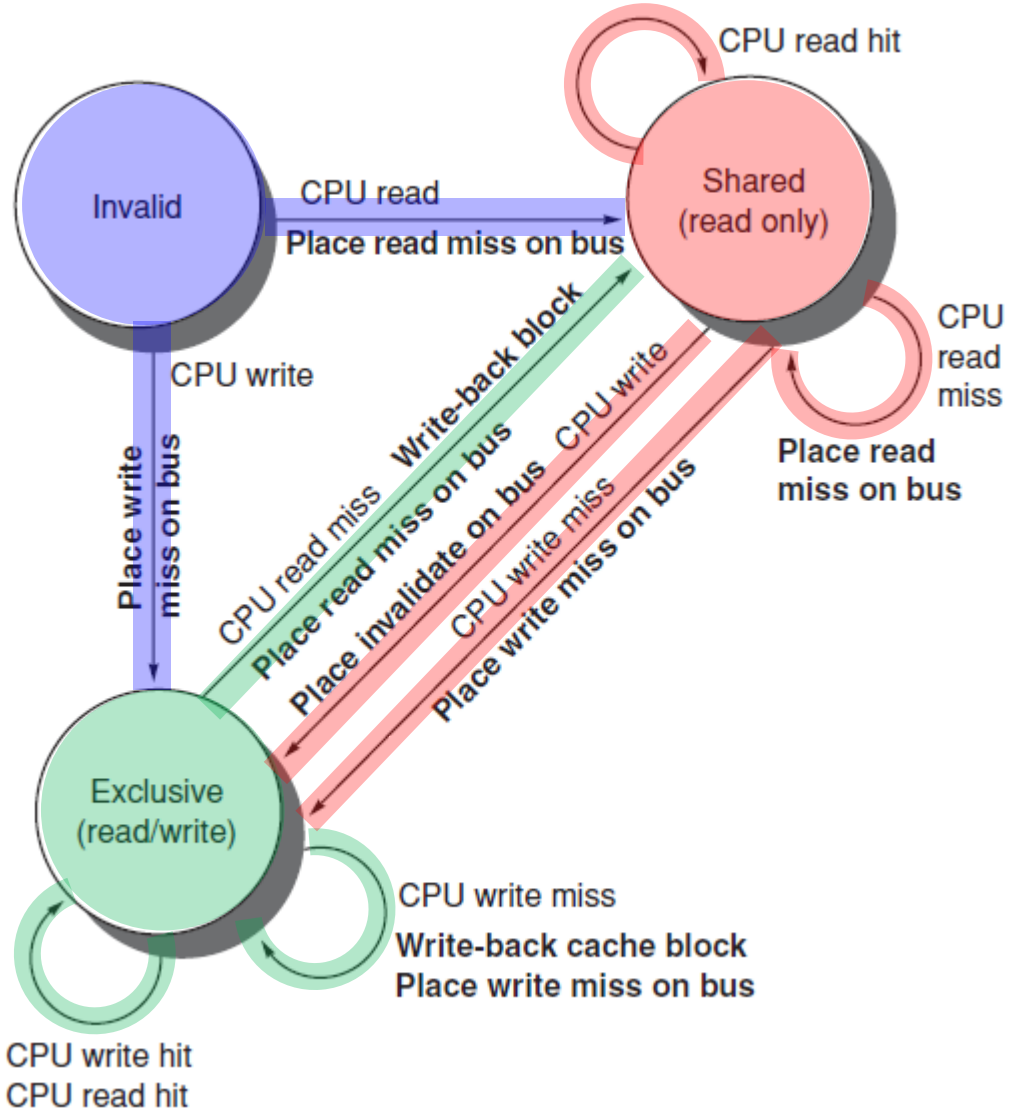
A snooping coherence protocol is implemented by incorporating a finite state controller in each node.

The controller responds to requests from the processor and the bus, changing the state of the selected block and using the bus to access data or to invalidate it.

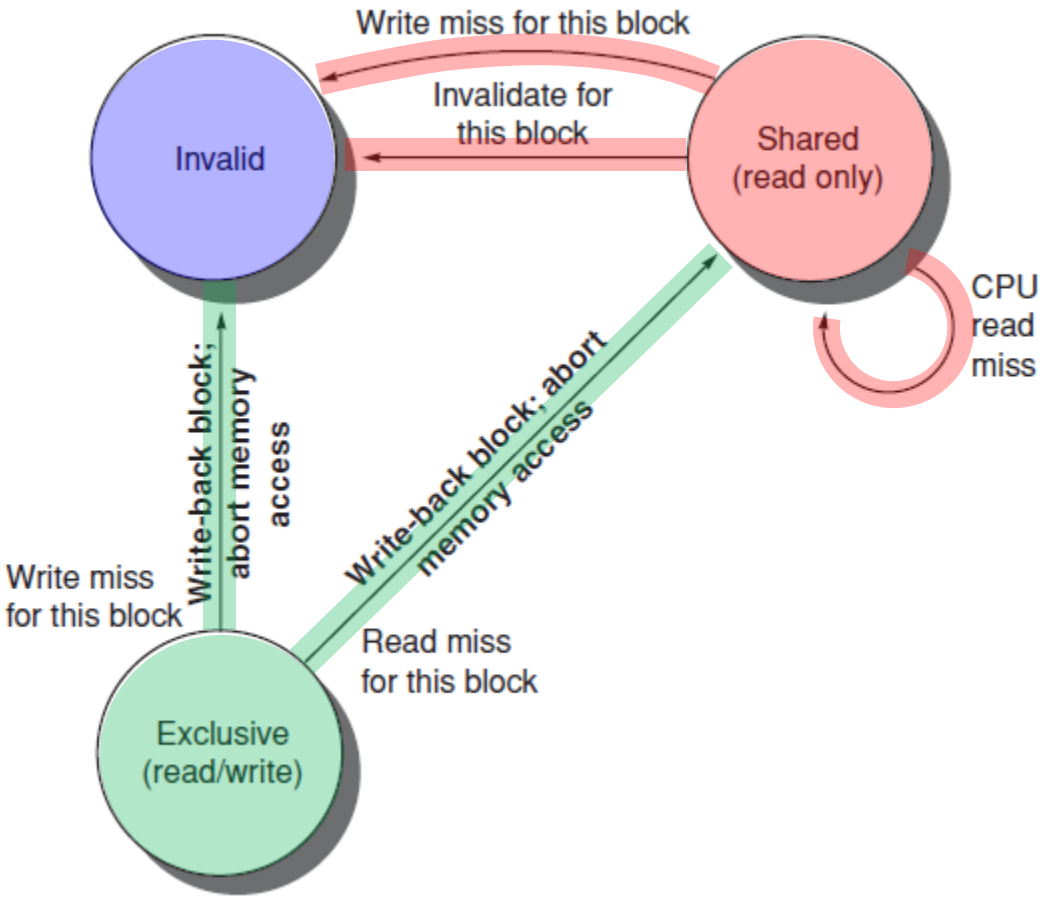
The protocol has three states: **invalid**, **shared**, and **modified**.

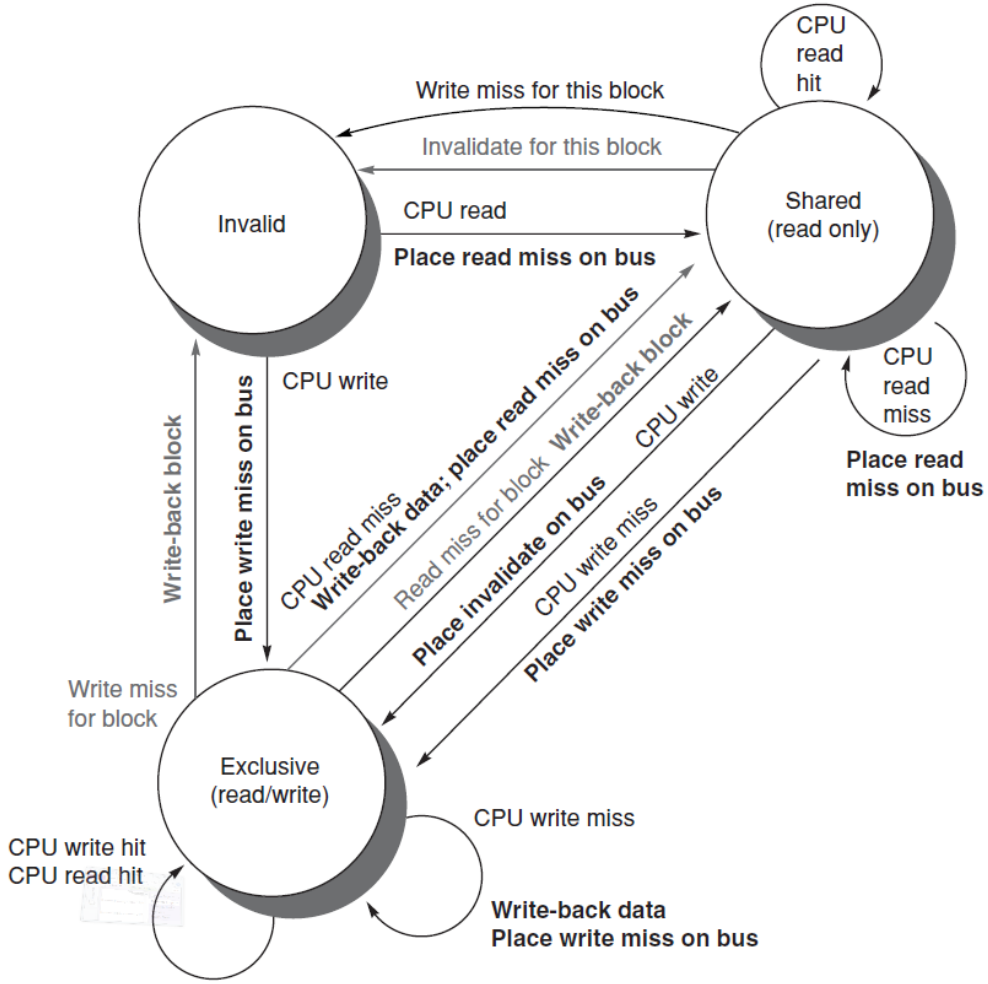
The **shared** state indicates that the block is potentially shared. The **modified** state indicates that the block has been updated in the cache, implying that it is **exclusive**.

Cache block state transitions for CPU requests



Cache block state transitions for bus requests





This protocol is for a write-back cache but is easily modified for a write through cache.

The normal cache **tags** can be used to implement the snooping and the **valid bit** to implement invalidation.

Read misses, whether generated by an invalidation or by some other event, are also straightforward since they simply rely on the snooping capability.

Writes like to know whether any other copies of the block are cached because otherwise the write need not be placed on the bus in a write-back cache.

Not sending the write on the bus reduces both the time taken by the write and the required bandwidth.

A bit indicating whether the block is shared enables to decide whether a write must generate an invalidate.

The cache generates an **invalidation** on the bus and marks the block as **exclusive**.

The processor with the sole copy of a cache block is normally called the **owner** of the cache block.

Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared.

Shared Memory Example

Assumes A1 and A2 map to same cache block

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2							WrMs	P2	A2			10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Performance of SMP (Shared Memory)

Misses arising from inter-processor communication, called **coherence misses**, are broken into **two sources**.

The first is **true sharing misses**, arising from data communication through **cache coherence** mechanism.

In an **invalidation-based** protocol, the first **write** by a processor to a **shared** block causes an **invalidation** to establish ownership of that block.

Additionally, when **another** processor attempts to **read** a modified word in that cache block, a miss occurs and the resultant block is transferred.

The second effect, called **false sharing**, occurs when a block is invalidated (subsequent reference causes a miss) because some word in the block, **other than** the one being read, is written into (at another processor).

If the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss, then it is a **false sharing miss**.

In a false sharing miss, the block is shared, but no word in the cache is actually shared. The **miss is always true if the block size were a single word**.

Example: The words $x1$ and $x2$ are in the same cache block, which is in the **shared** state in the caches of both **P1** and **P2**.

For the following sequence of events, identify each miss as a true or a false sharing miss, or a hit.

Any miss would be true sharing miss if the block size were one word.

1. **True** sharing miss, since $x1$ is in **P2** and needs to be invalidated from **P2**.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

2. **False** sharing miss, since $x2$ was invalidated by the write of $x1$ in **P1**, but that value of $x1$ is not used in **P2**.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

3. **False** sharing miss, since the block containing $x1$ is marked **shared** due to the read in **P2**, but **P2** did not read $x1$.

The cache block containing $x1$ will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block.

4. **False** sharing miss for the same reason as step 3.

5. **True** sharing miss, since the value being read was written by **P2**.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

The role of coherence misses is more significant for tightly coupled applications that share significant amounts of user data.