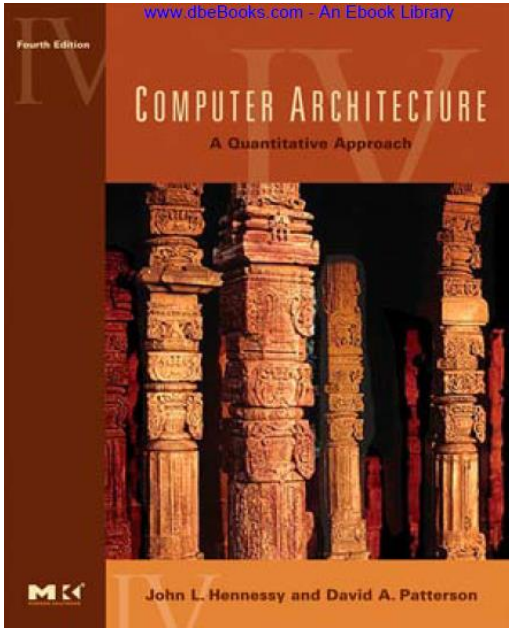
Multithreading and Multiprocessing

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



ILP Limitations

Pipelining, multiple issue, dynamic scheduling and speculation have limits.

Since late 90's focus turned to clock speedup, without increasing issue rates.

In 2010 clock speedup ended too.

Multithreading is an addition to ILP.

Multiprocessing: Driven by huge silicon integration technology, data processing explosion, internet, cloud computing, ...

Example. Compare three CPUs running on benchmark achieving ideal issue rate as shown.

1. 2.0 GHz two-issue static-pipe simple MIPS, achieving 0.6 CPI. Its cache yields 1.0 % misses/Instruction.

2. 5.0 GHz single-issue with deep static pipe MIPS, achieving 1.2 CPI . Its larger cache yields 0.5% misses/Instruction.

3. 1.0 GHz four-issue speculative superscalar MIPS. Its smallest cache yields 1.5% misses/instruction, but its dynamic scheduling hides 50% of the miss penalty.

Cache miss penalty is 100 nSec in all processors.

Determine the relative **MIPS** (10^6 *instructions/sec*) performance of these three processors.

Answer. Miss rate is first used to compute the contribution to CPI from cache misses.

$$\text{Cache Miss CPI} = \text{Misses per instruction} \times \text{Miss penaly}$$

$$\text{Miss penalty} = \text{Memory access time}/\text{Clock cycle}$$

$$\text{Miss penalty}_1 = 100 \text{ nSec}/0.5 \text{ nSec} = 200 \text{ cycles}$$

$$\text{Miss penalty}_2 = 100 \text{ nSec}/0.2 \text{ nSec} = 500 \text{ cycles}$$

$$\text{Miss penalty}_3 = 0.5 \times 100 \text{ nSec}/1.0 \text{ nSec} = 50 \text{ cycles}$$

Combine Miss penalty and miss rate for CPI penalty.

$$\text{Cache Miss CPI}_1 = 0.01 \times 200 = 2.0$$

$$\text{Cache Miss CPI}_2 = 0.005 \times 500 = 2.5$$

$$\text{Cache Miss CPI}_3 = 0.015 \times 50 = 0.75$$

CPI is known for the first two CPUs. For 3rd, there is

$$\text{Pipeline CPI}_3 = 1/\text{Issue rate} = 1/4.0 = 0.25$$

CPI is the sum of the pipeline and cache miss CPIs

$$\text{CPI}_1 = 0.6 + 2.0 = 2.6$$

$$\text{CPI}_2 = 1.2 + 2.5 = 3.7$$

$$\text{CPI}_3 = 0.25 + 0.75 = 1.0$$

The three CPUs are compared in terms of their MIPS.

$$\text{Instr. execution rate} = \text{Clock rate}/\text{CPI}$$

$$\text{Instr. execution rate}_1 = 2000\text{MHz}/2.6 = 770\ \text{MIPS}$$

$$\text{Instr. execution rate}_2 = 5000\text{MHz}/3.7 = 1351\ \text{MIPS}$$

$$\text{Instr. execution rate}_3 = 1000\text{MHz}/1.0 = 1000\ \text{MIPS}$$

Multithreading

Significant parallelism occurring naturally at higher levels in apps cannot be exploited by ILP techniques.

Multithreading (**MT**) tolerates or masks long and often unpredictable latency operations by **switching to another context**, which is able to do a useful work.

Such higher-level parallelism is called **Thread-Level Parallelism** (**TLP**).

A **threads** is a separate process with its own instructions, data, PC, and registers, must for execution.


TLP and **ILP** exploit two **different** kinds of parallel structure in a program.

FUs of a CPU designed to exploit ILP are often idle (stalls, dependences).

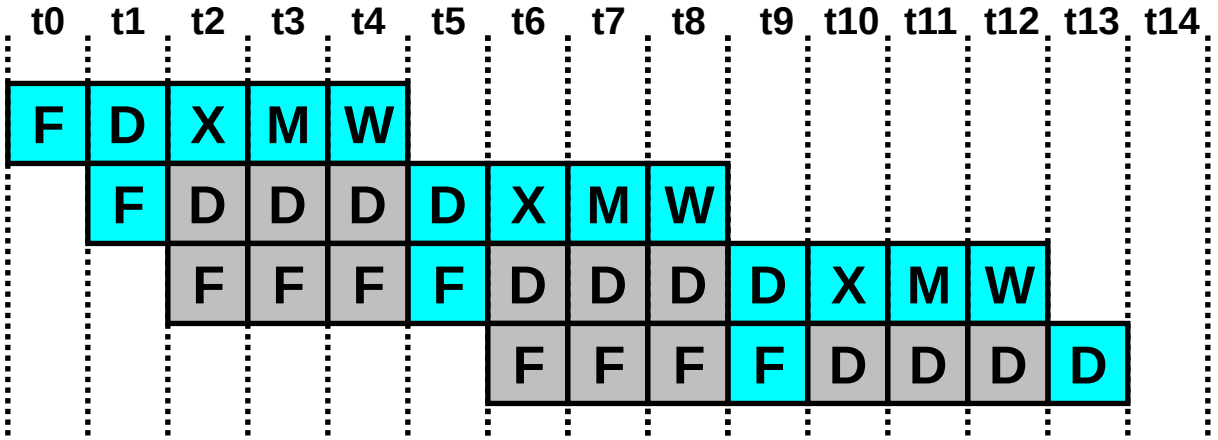
Question: Can **ILP**-designed processor exploit **TLP**?

MT allows multiple threads sharing **FU**s of a single **CPU**.

Memory is shared through the **VM**, already supporting **multiprogramming**.

CPU must support **quick thread switching**, much faster than process switch taking 100s-1000s CPU cycles.

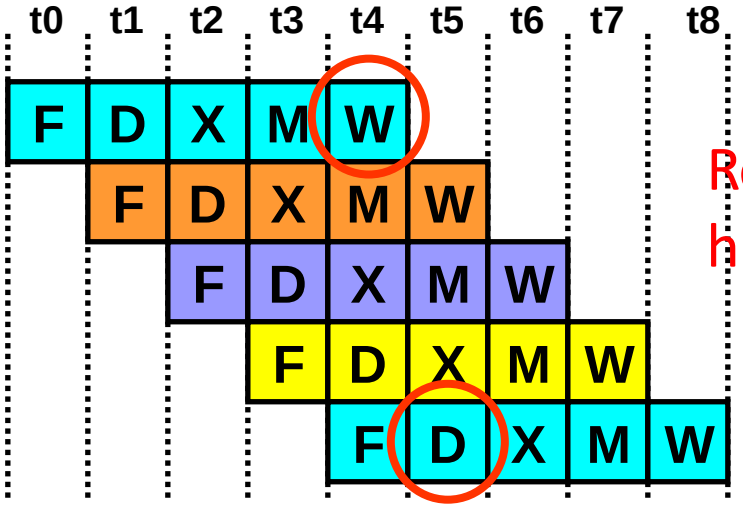
LW r1, 0(r2)
LW r5, 12(r1)
ADDI r5, r5, #12
SW r5, 12(r1)



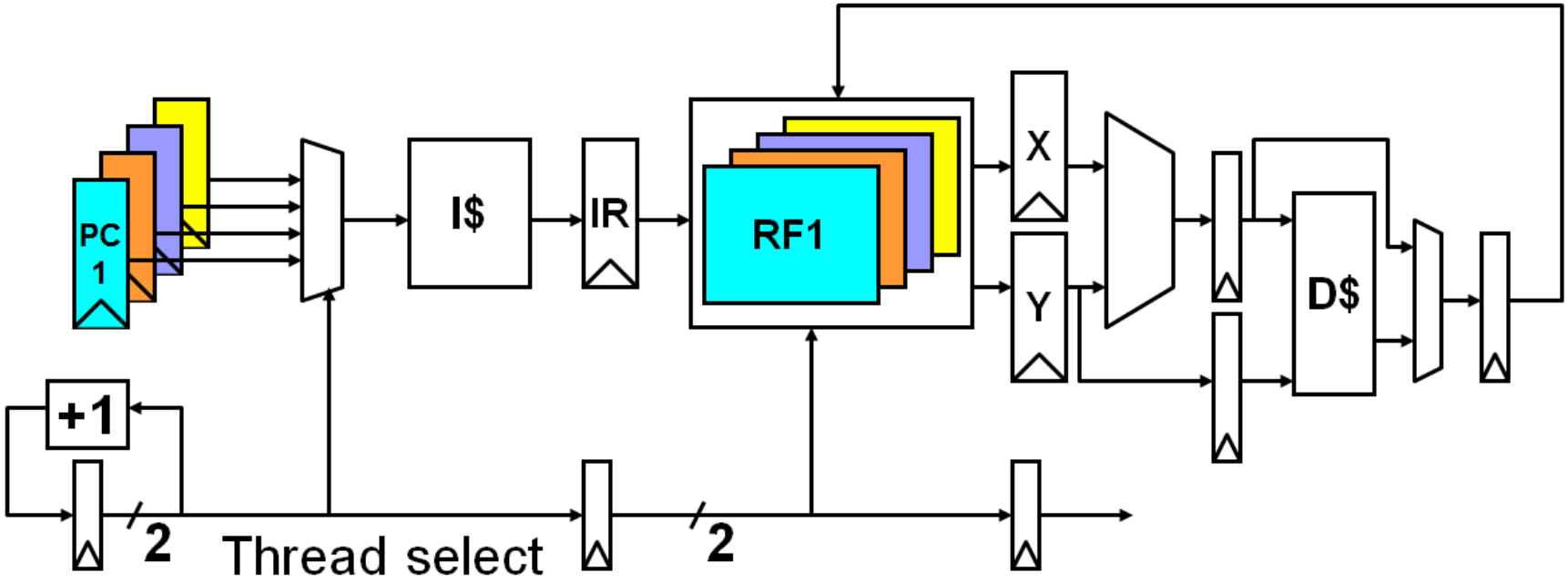
Each instruction may depend on the previous.

Interleave 4 threads on non-bypassed 5-stage pipe

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW r5, 0(r7) ???
T1: LW r5, 12(r1)



Resolve RAW hazards



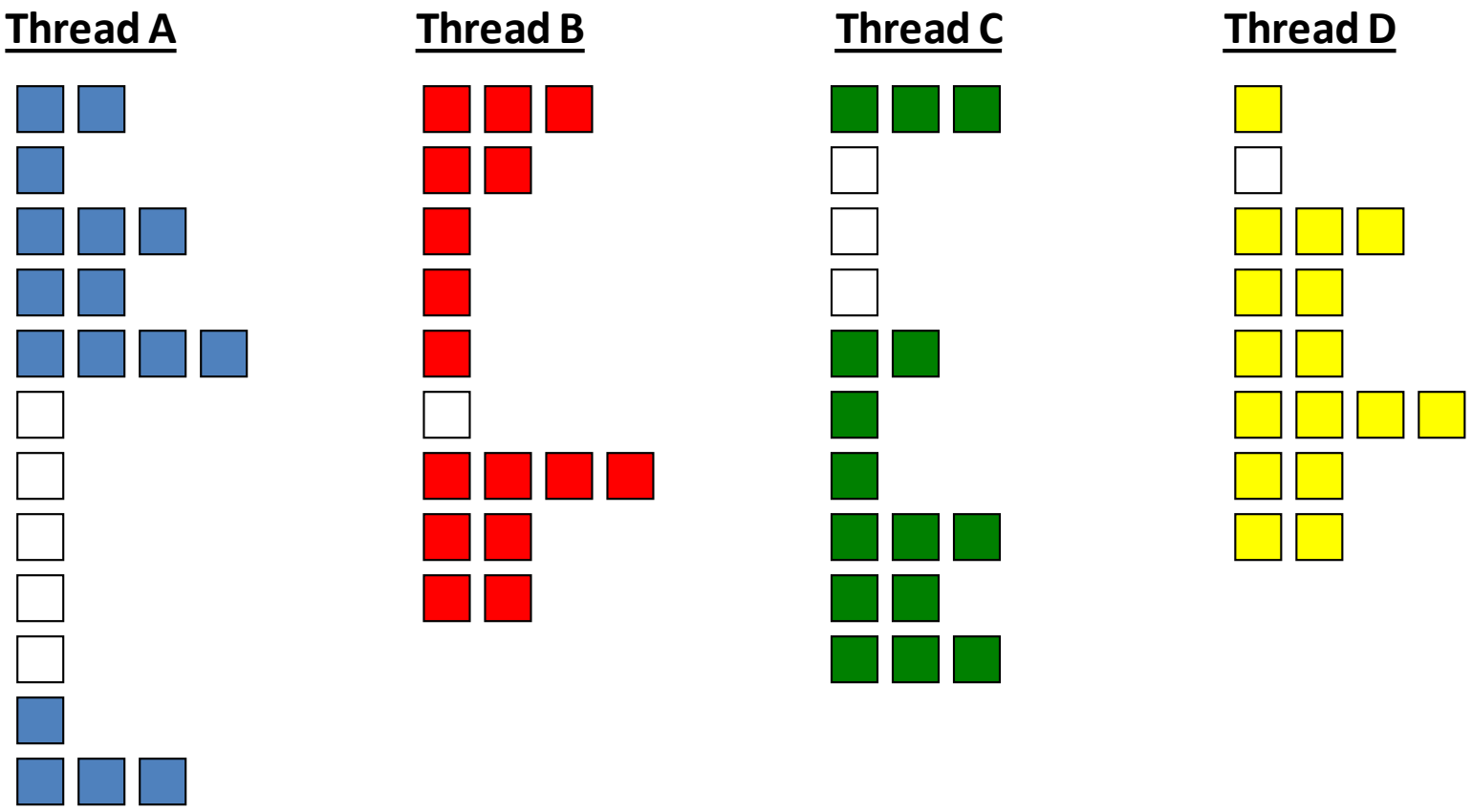
Carry thread select down pipeline to ensure correct state bits read/written at each pipe stage.

Appears to SW (OS) as multiple, albeit slower, CPUs.

Design challenges: cache miss increases, slower RF.

MT Approaches

Example: 4-Issue Machine




Two approaches: **Fine-grained** and **Coarse-grained**.

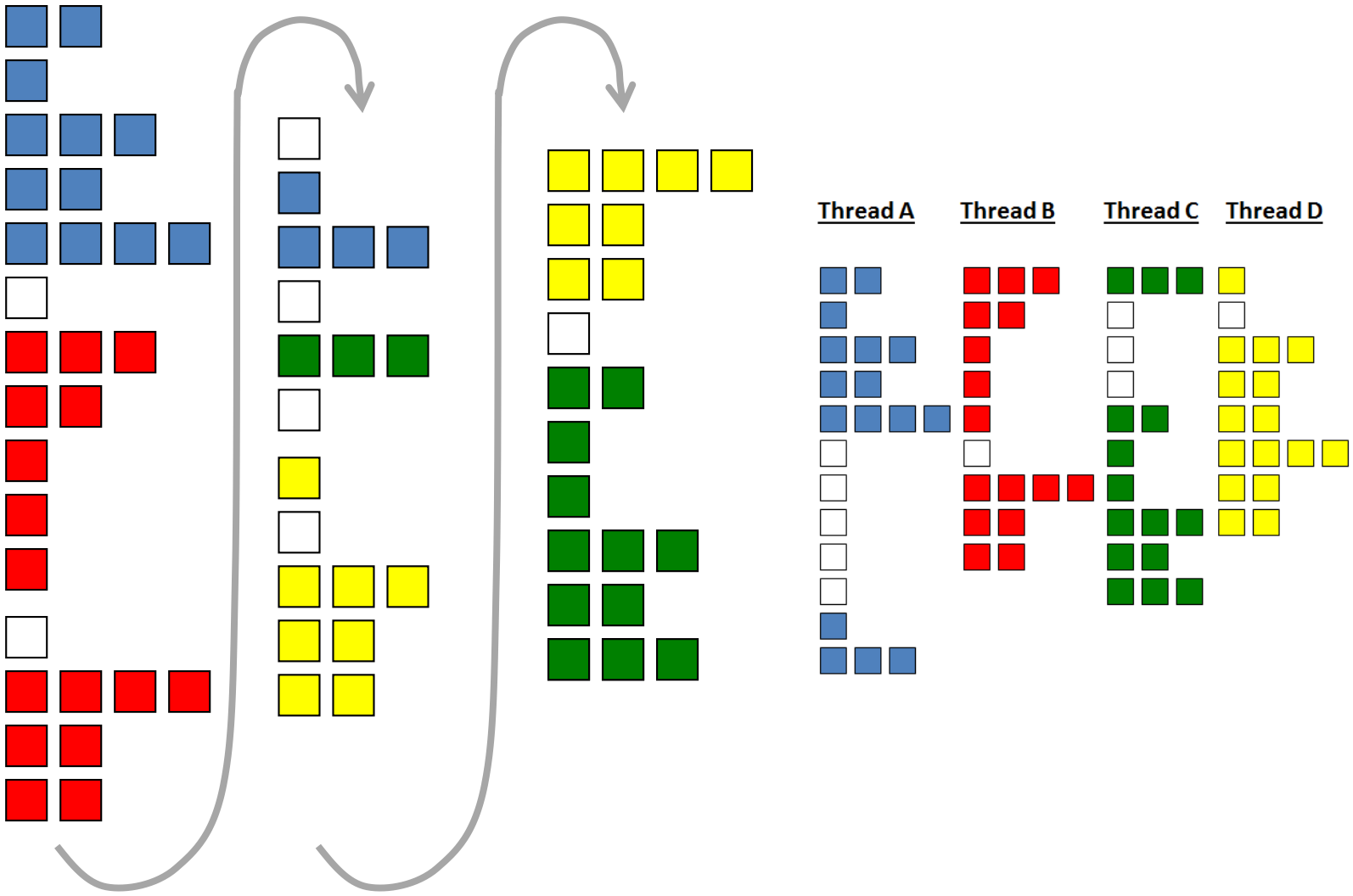
Coarse-grained MT switches threads only on costly stalls (L2 misses), where pipeline refill is negligible compared to the stall time.

CPU is hardly slowed down by thread switching.

Cannot exploit short stalls (employment of FUs).

Start-up overhead: thread switching must empty pipeline. New thread must fill it before instructions will complete.

Coarse-grained MT



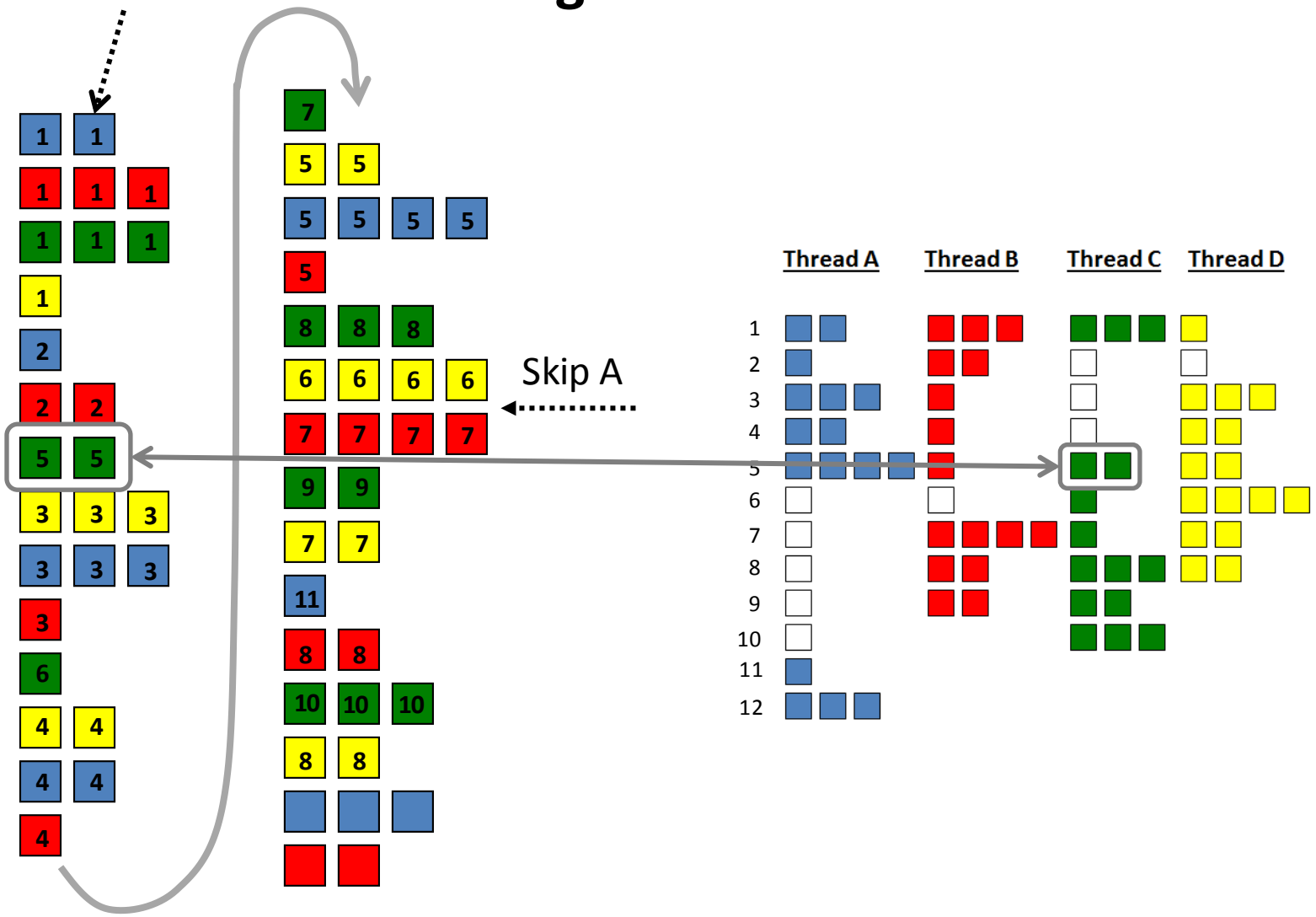
Fine-grained MT switches between threads on each instruction (every cycle), multiple threads interleave.

Round-robin thread switching, skipping any stalled threads.

Advantage: hiding throughput losses arising from short stalls, instructions of other threads are executed when other are stalled.

Disadvantage: Individual threads execution slowdown, thread ready to execute without stalls is delayed by other threads.

Fine-grained MT



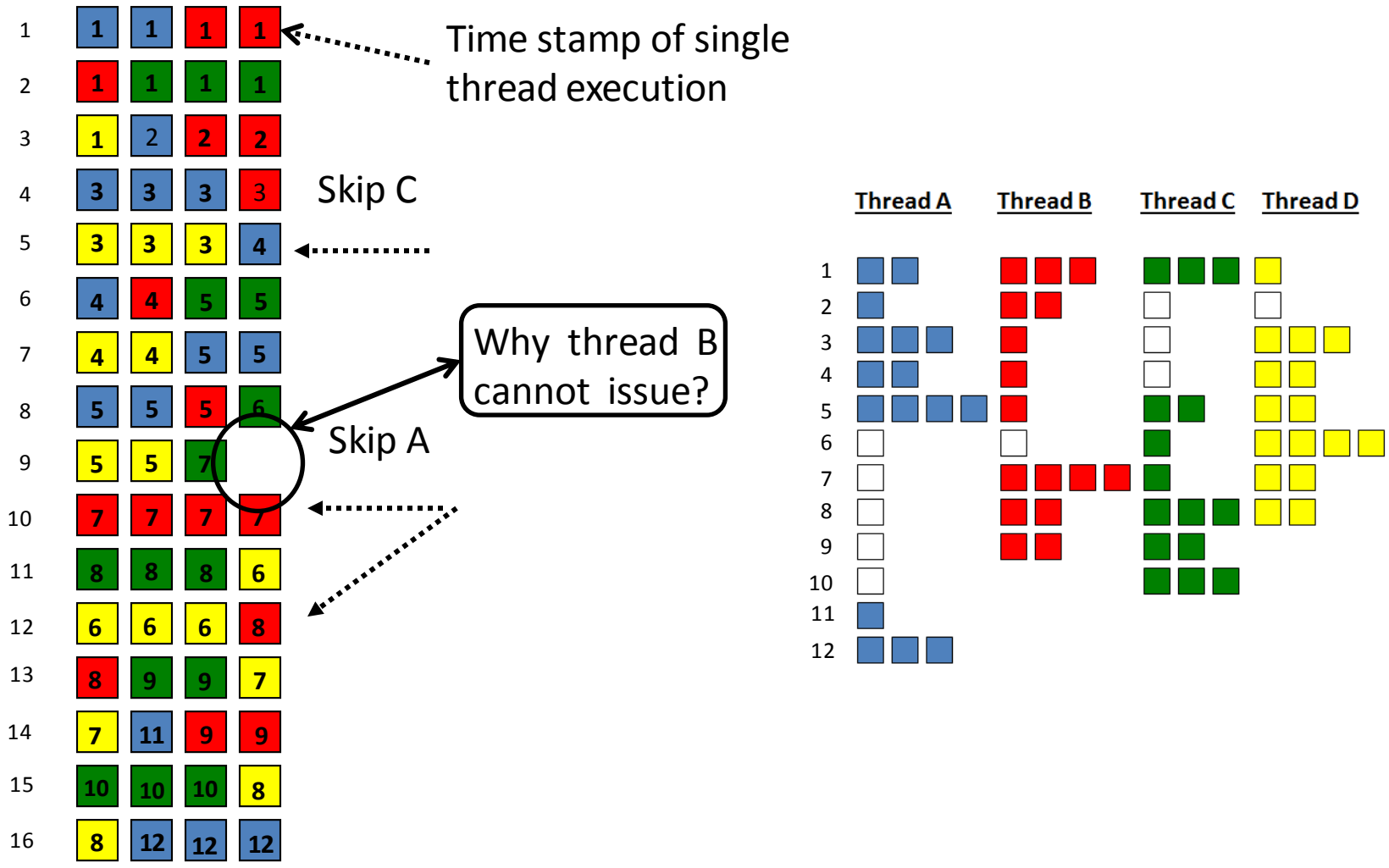
Simultaneous Multithreading

Simultaneous multithreading (**SMT**) is a variation on MT to exploit **TLP** simultaneously with **ILP**.

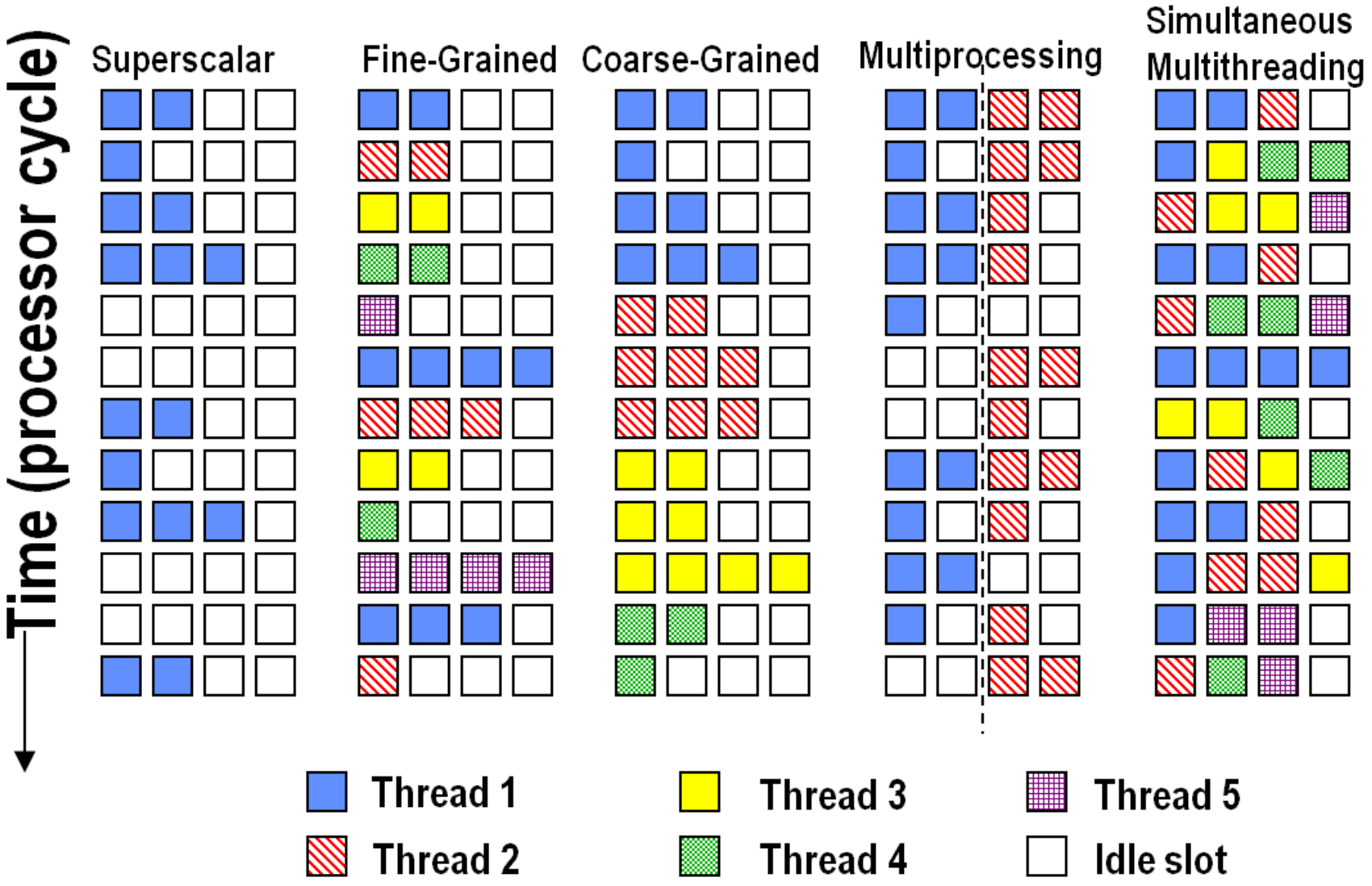
Motivated by **multiple-issue** CPUs which have more **FU** parallelism than a single thread can effectively use.

Register renaming and dynamic scheduling enables issuing multiple instructions from **independent threads ASAP** regardless of the **dependences** among them.

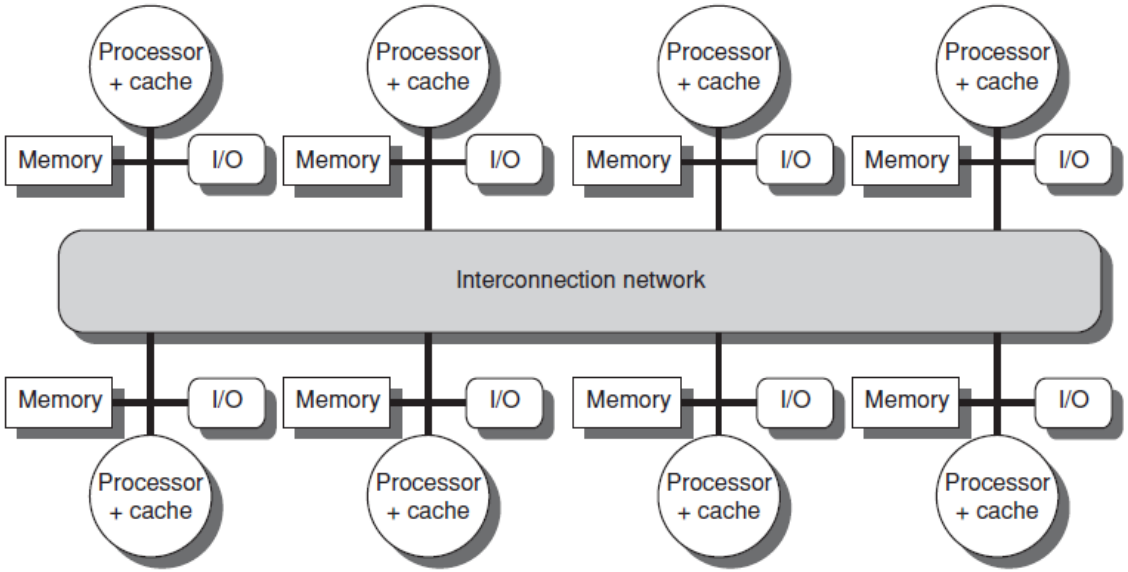
Dependences resolution can be handled by the dynamic scheduling capability (**Tomasulo, ROB**).


1 1 1 1
1 1 1 1
1 2 2 2
3 3 3 3
3 3 3 4
4 4 5 5
4 4 5 5
5 5 5 6
5 5 7
7 7 7 7
8 8 8 6
6 6 6 8
8 9 9 7
7 11 9 9
10 10 10 8
8 12 12 12
Time stamp of single thread execution
Skip C
Why thread B cannot issue?
Skip A
Thread A
Thread B
Thread C
Thread D

Approaches to use the issue slots.



Distributed Shared Memory (DSM)



Many CPUs, memory is distributed among CPUs rather than centralized. Otherwise long latency occurs.

Cost-effective to scale memory bandwidth if most accesses are to local memory. Complex communication.

Communication occurs through shared address space via load and store operations.

Physically separate memories are addressed as **one logical** address space.

Memory reference made by any processor to any address.

Speedups is limited by **parallelism** in programs. **High communications cost**, 50 ÷ 1000 clock cycles.

Example. What fraction of original computation must be parallel to achieve 80X speedup with 100 CPUs?

Program operates in two modes:

- parallel - all processors (100) fully used or
- serial - only one processor in use.

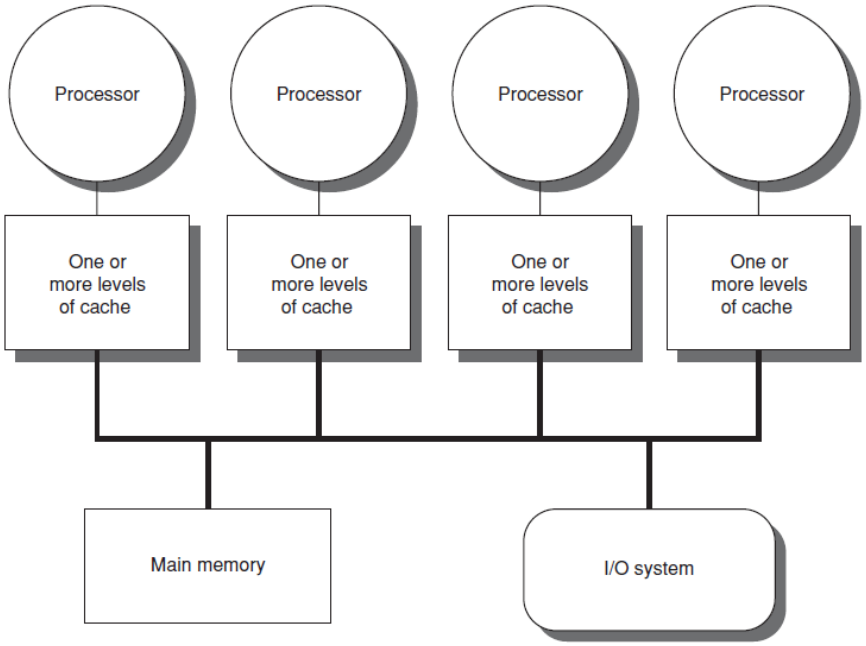
$$\text{total speedup} = \frac{1}{\frac{\text{parallel}}{\text{speedup parallel}} + (1 - \text{parallel})}$$

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

$\text{parallel} = 0.9975$! (less than 0.25% serial code!)

For linear speedup the entire must have no serial portions. Not realistic. ■

Centralized Shared-Memory (SMP)



Single memory (address space) satisfies the demands of few CPUs.

Called Symmetric (shared-memory) **Multiprocessors** (**SMPs**).

SMPs support caching of both:

Private data used by a single CPU.

shared data used by all CPU, providing communication through reads and writes of the shared data.

For **private** item caching the program behaves as uniprocessor since no other CPU uses the data.

For **shared** data caching the shared value may be replicated in multiple caches.

New problem: **cache coherence**. Different CPUs may see different values for same address.

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.

Expect also in uniprocessors.

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

   If insufficiently separated, X may not left P_1 yet.

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

When exactly a written value must be seen by a reader is defined by **memory consistency model**.

Assume **write does not complete** (not allowing next write) until all CPUs see the effect of that write.

Assume CPU **not changing order** of any write w.r.t any other memory access.

Above implies that if CPU writes address A followed by address B, any CPU seeing new B must also see new A.

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**.

Snooping protocols have no centralized state.

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

All caches are accessible via a bus or switch.

All cache controllers **snoop** (monitor) to determine whether they own a copy of block 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.

Consider a write followed by a read by another CPU.

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

Write Invalidate Implementation

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

The CPUs 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.


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.

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 CPU 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

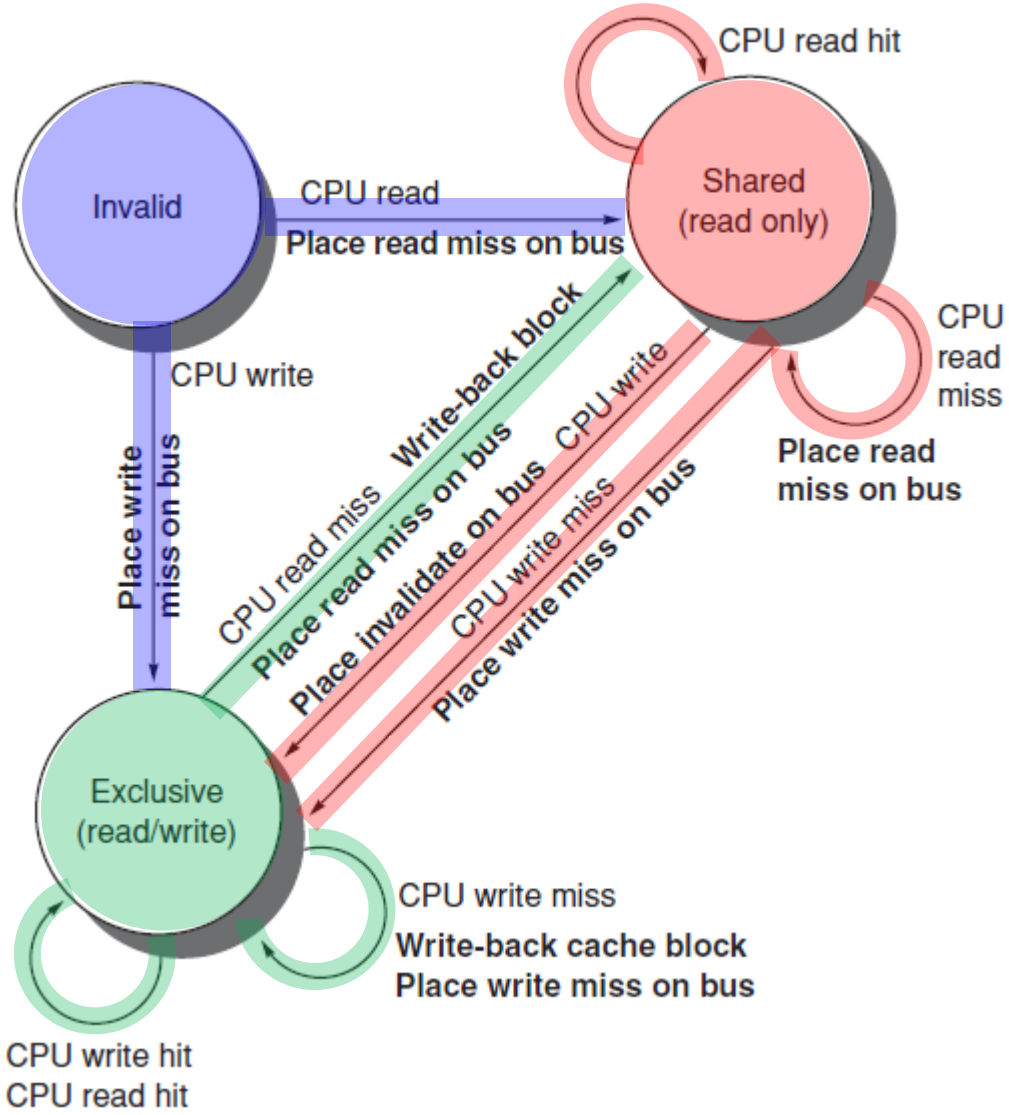
A snooping coherence protocol is implemented by incorporating a finite state controller in each node.

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.

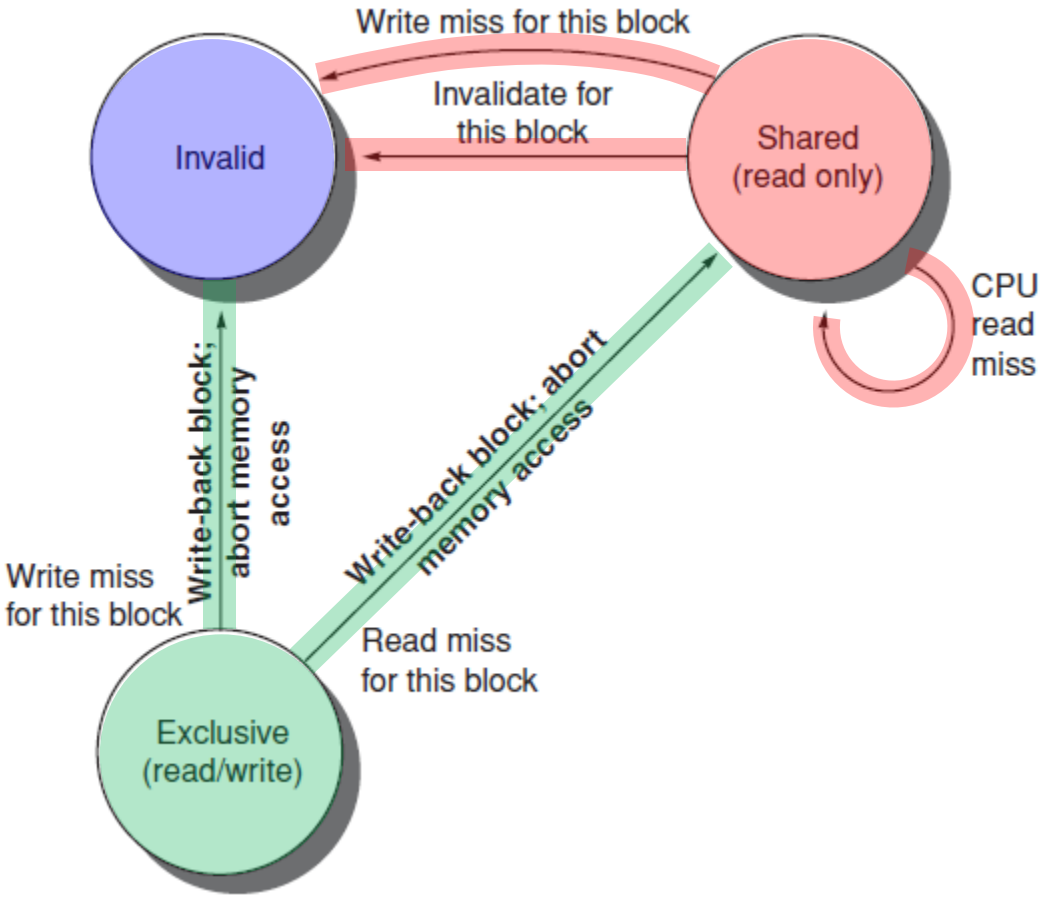
The protocol has three states: **invalid**, **shared**, and **modified**.

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.

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