National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 14: Multicores

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - 6.888 at MIT
    - https://courses.csail.mit.edu/6.888/spring13/
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2024/

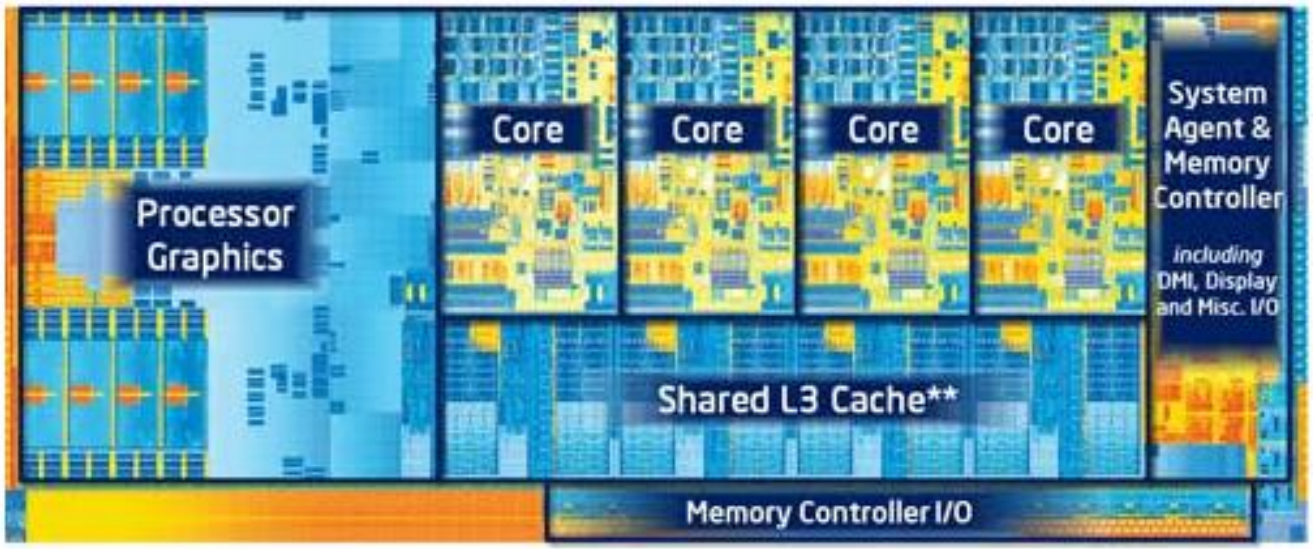
Outline

- Multi-core Processor
- Hardware Multi-Threading
- Cache Coherence
- Vector Processors
- Graphics Processors

Multiplying Performance

- A single core can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
- What if we need even more computing power?
  - Use multiple cores! But how?
- Old-school (2000s): Ultra Enterprise 25k
  - 72 dual-core UltraSPARC IV+ processors
  - Up to 1TB of memory
  - Niche: large database servers
  - $$$, weighs more than 1 ton
- Today: multicore is everywhere
  - Can't buy a single-core smartphone…
    - …or even smart watch!

Intel Quad-Core “Core i7”



Application Domains for Multicore

- Scientific computing/supercomputing
  - Examples: weather simulation, aerodynamics, protein folding
  - Large grids, integrating changes over time
  - Each core computes for a part of the grid
- Server workloads
  - Example: airline reservation database
  - Many concurrent updates, searches, lookups, queries
  - Each core handle different requests
- Video games
  - Each core renders part of a frame

But software must be written to expose parallelism

Multicore is Energy Efficient

- Explicit parallelism (multicore) is highly energy efficient
- Recall: dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel's Xscale
    - 1 GHz → 200 MHz reduces energy used by 30x
- Consider the impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th
- And, amortizes background "uncore" energy among cores
- Assumes parallel speedup (a difficult task)
  - Subject to Amdahl's law

Amdahl's Law

- Restatement of the law of diminishing returns
  - Total speedup limited by non-accelerated piece
  - Analogy: drive to work & park car, walk to building
- Consider a task with a "parallel" and "serial" portion
  - What is the speedup with N cores?
  - Speedup(n, p, s) = (s+p) / (s + (p/n))
    - p is "parallel percentage", s is "serial percentage"
  - What about infinite cores?
    - Speedup(p, s) = (s+p) / s = 1 / s
- Example: can optimize 50% of program A
  - Even a "magic" optimization that makes this 50% disappear…
  - …only yields a 2X speedup

Uniprocessor Concurrency

- **A software thread**
  - Private per-thread PC, registers, stack
  - Shared state: memory (globals, heap)
    - actually stack lives in memory so it can be shared, too
  - Threads share the same virtual memory address space
    - A process is like a thread, but with its own address space
  - Languages (Java, C, C++, …) have built-in threading support
- the Operating System (OS) manages threads
  - In single-core system, all threads share one processor
    - Hardware timer interrupt occasionally activates OS
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in an OS course!

Shared Memory Programming Model

- Programmer explicitly creates multiple threads
- Loads & stores access a single **shared memory**
  - All memory is shared & accessible by all threads
- A thread switch can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send/receive network messages)
  - **Express parallel work via Thread-Level Parallelism**
    - This is our focus!

Shared Memory Model: Interleaving

- Initially: all variables zero (that is, x=0, y=0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

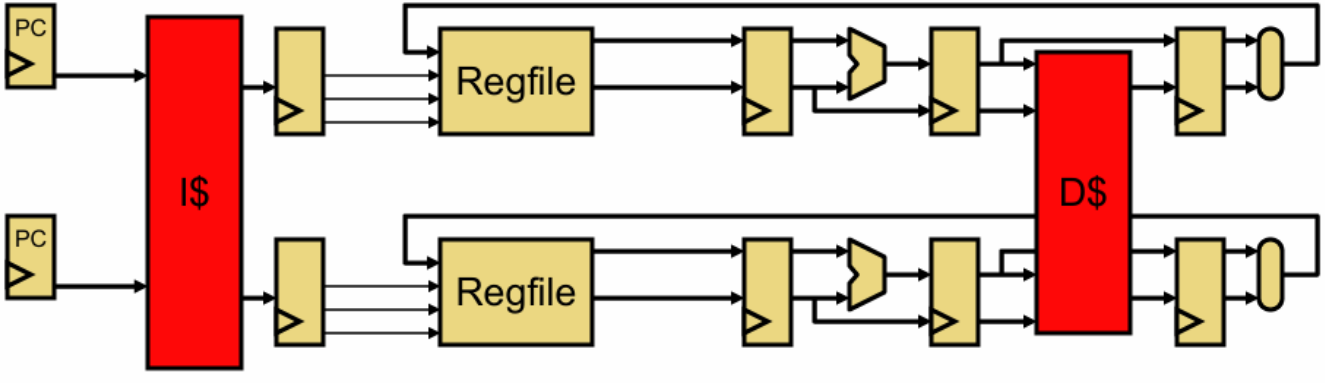
store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0, y=0)?

Shared Memory Implementations

- **Multiplexed uniprocessor**
  - Runtime system and/or OS pre-empt & swap threads
  - Interleaved, but no parallelism
- **Multiprocessors**
  - Allow private caches, much better performance
  - Same interleaved shared-memory model
- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model
- **All have the same programming model**

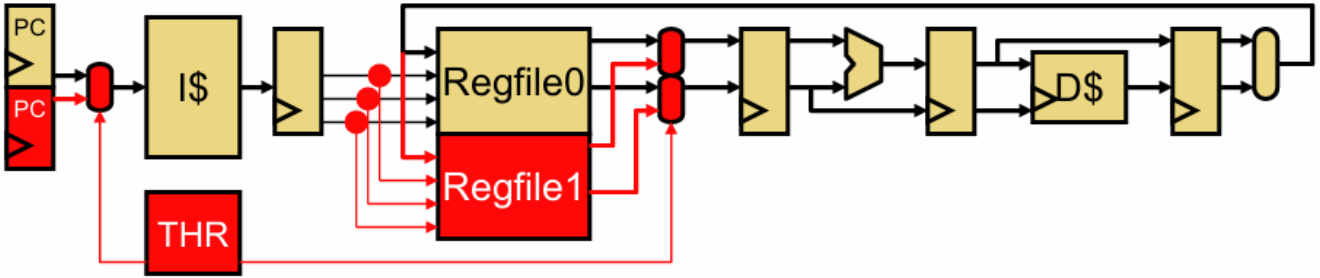
Simplest Multiprocessor



- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Share the caches for now (but we'll fix this soon)
- Can execute multiple threads
  - Valid implementation of shared memory programming model
  - Operations (loads and stores) are interleaved "at random"
  - Loads returns the value written by most recent store

Hardware Multithreading

- **Not** the same as software multithreading!
- Each **hardware thread** can run one sw thread
  - threads can come from same or different processes



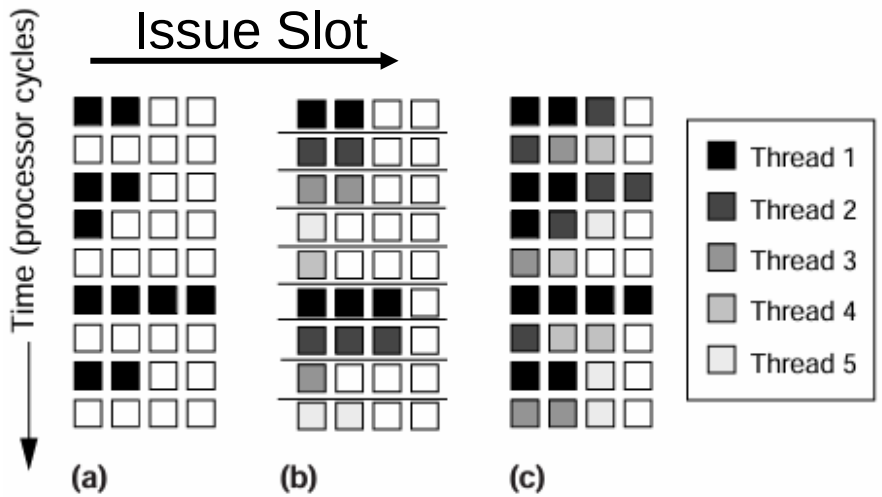
- **Hardware Multithreading (MT)**
  - Multiple hardware threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions

Hardware Multithreading

- Why use hw multithreading?
  - + **Multithreading improves utilization and throughput**
    - Single programs underutilize pipeline (e.g., cache miss)
    - allow insns from different hw threads in pipeline at once
  - **Multithreading does not improve single-thread performance**
    - Individual threads may run slower
  - **Coarse-grain MT**: switch on cache misses: why?
  - **Simultaneous MT**: fine-grained switching
    - Intel calls this *hyperthreading*

Hardware Multithreading

- **Hardware Multithreading**
  - (a) Superscalar
  - (b) Fine-grain MT superscalar
  - (c) SMT processor
  - A filled box indicates the processor found an inst to execute in that issue slot on that cycle
  - Horizonal waste -> poor ILP
  - Vertical waste -> long latency inst (memory access) inhibits further inst issue

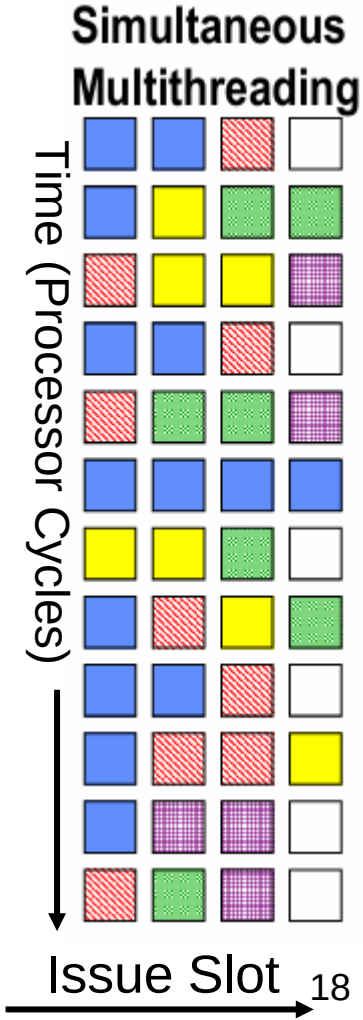


Hardware Multithreading

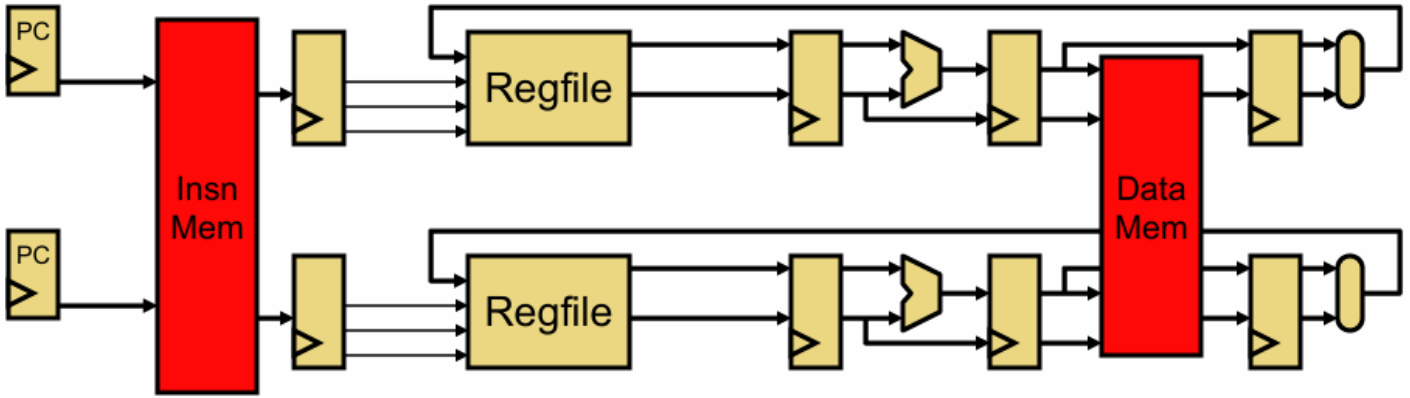
- **Hardware Multithreading**
  - On any given cycle a processor executes instructions from one of the threads
  - On the next cycle, it switches to a different thread context and executes instructions from the new thread
  - Better tolerate long latency operations
    - Remove vertical waste
    - Cannot remove horizontal waste as instruction issue width increases

Hardware Multi-Threading

- **Simultaneous multithreading (SMT)**
  - Exploiting TLP in a single processor core
  - Each clock, core chooses instructions from any threads that can issue
  - Dynamic schedules machine resources among instructions
  - Needs one context per thread
  - Benefits
    - Improve the hardware utilization
    - No partitioning of many resources
    - E.g. Intel Hyper-threading

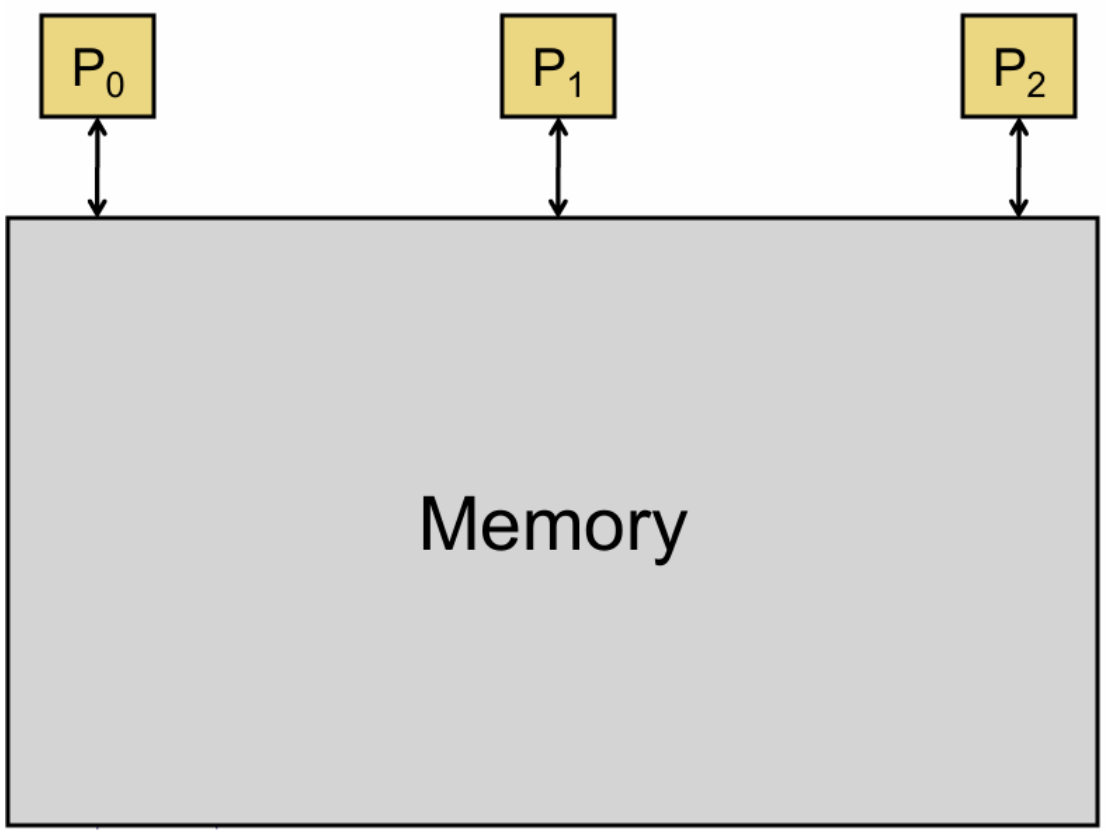


Simplest Multiprocessor

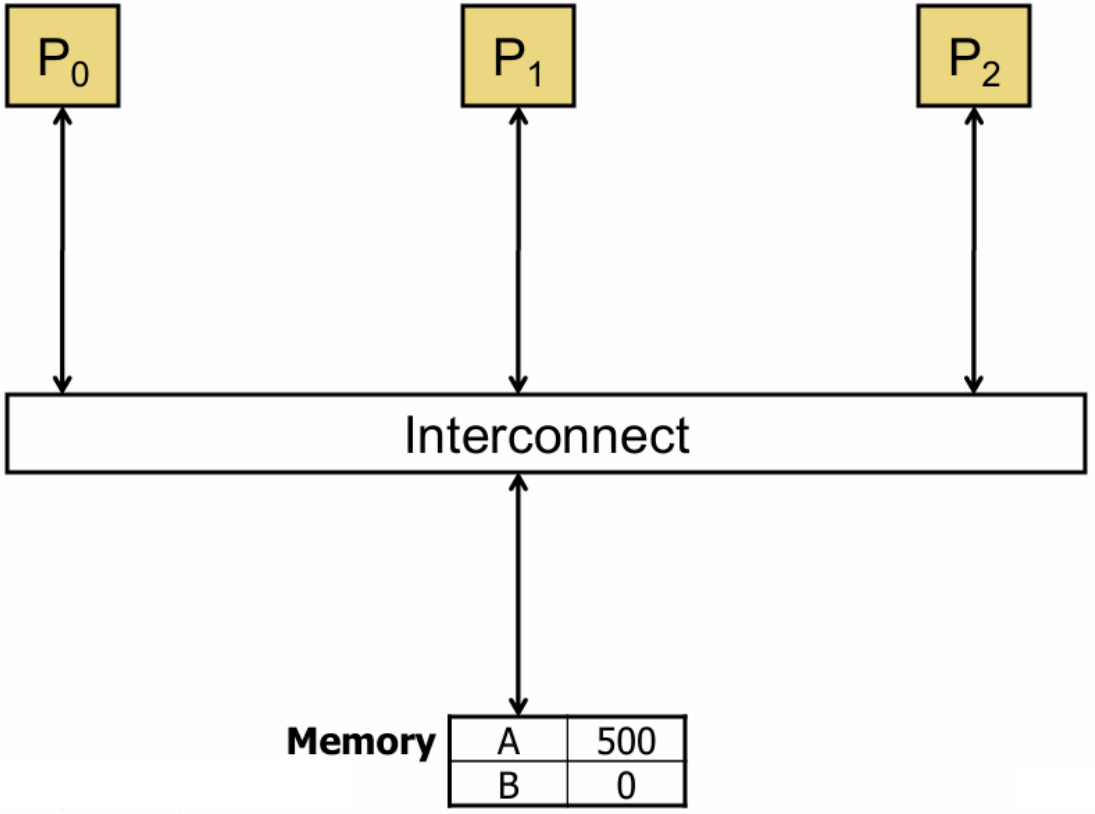


- Sharing L1 caches is slow
  - Hurts bandwidth and latency
- Solution: use per-processor ("private") caches
  - Coordinate them with a **Cache Coherence Protocol**
- Must still provide shared-memory invariant:
  - **Loads read the value written by the most recent store**

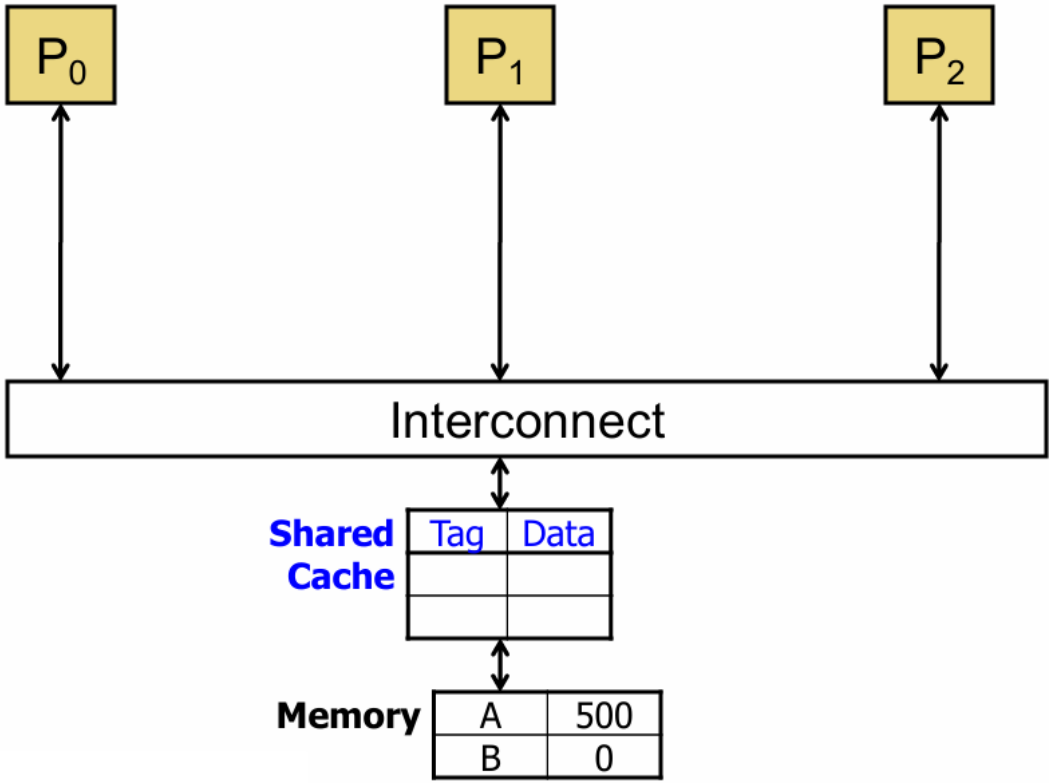
Starting point: no caches



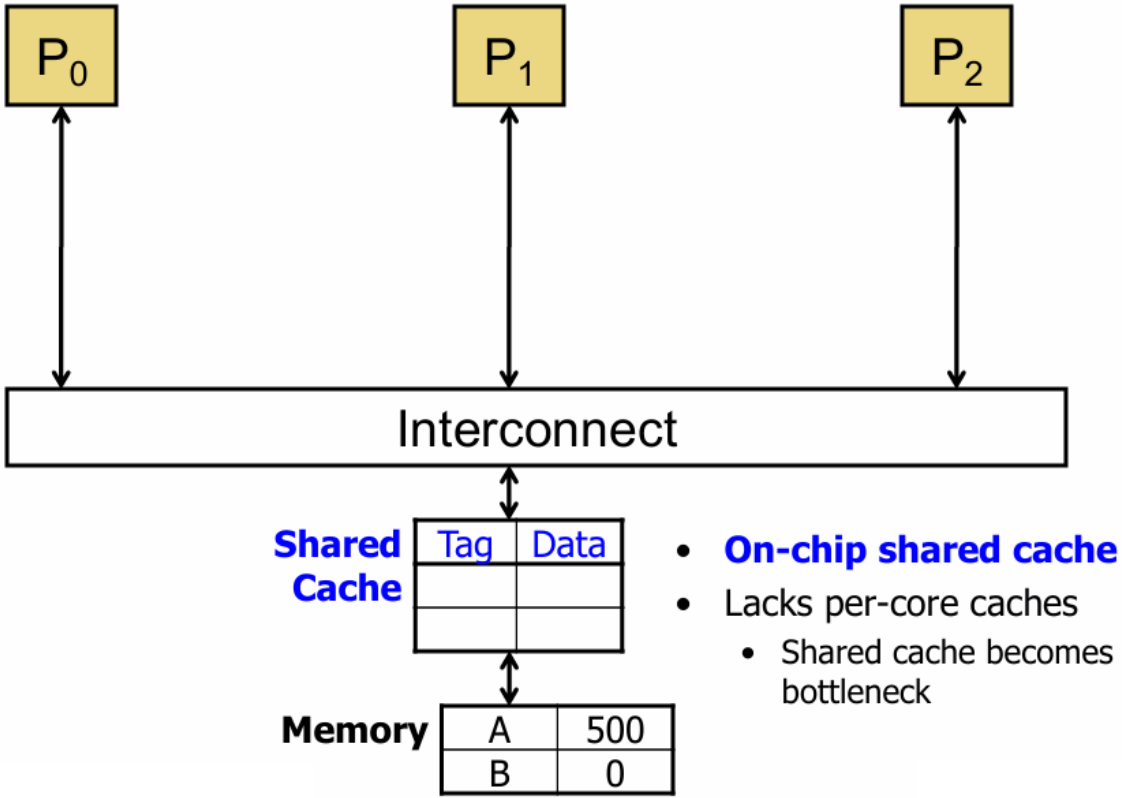
Starting point: no caches



Add a Shared Cache

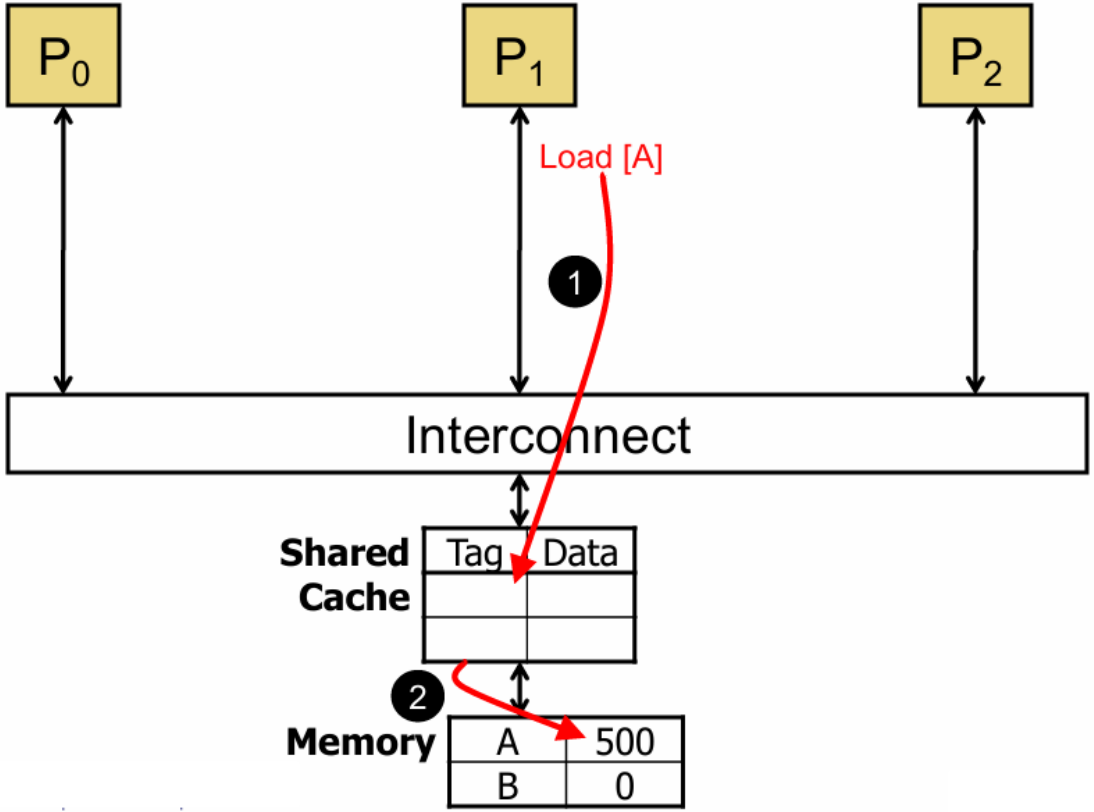


Shared Cache Implementation

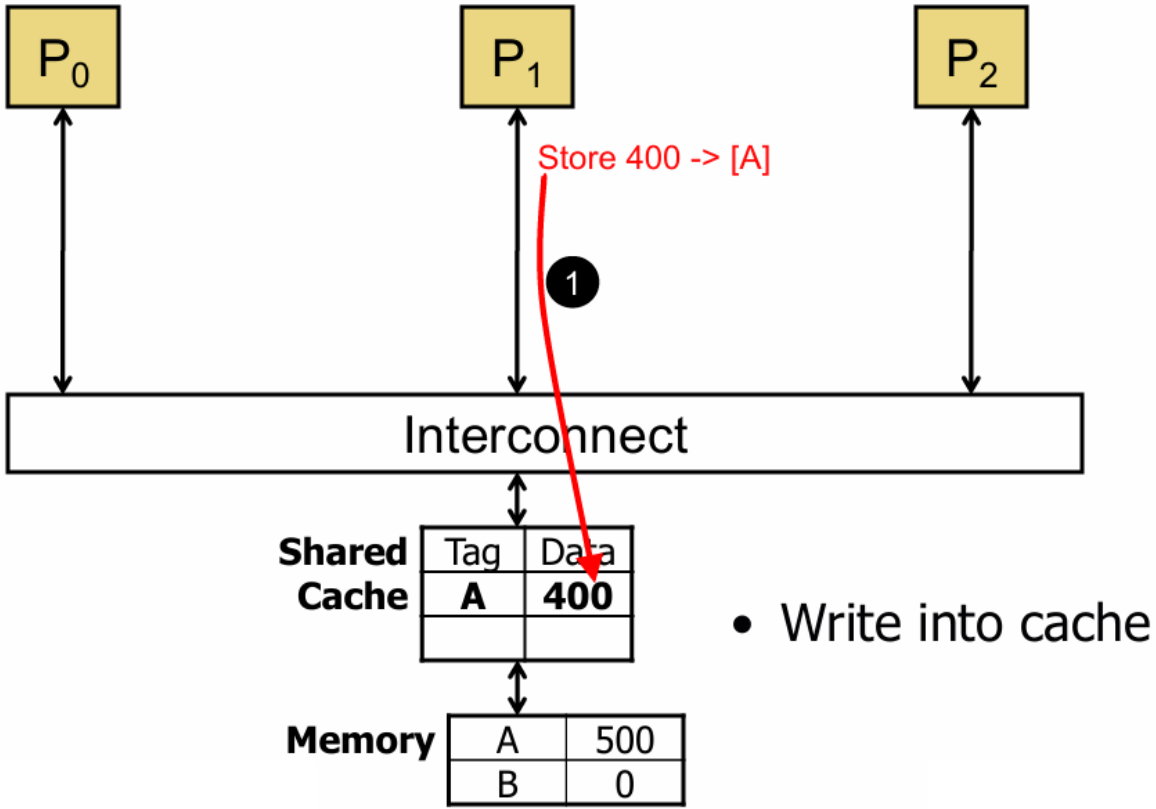


- **On-chip shared cache**
- Lacks per-core caches
  - Shared cache becomes bottleneck

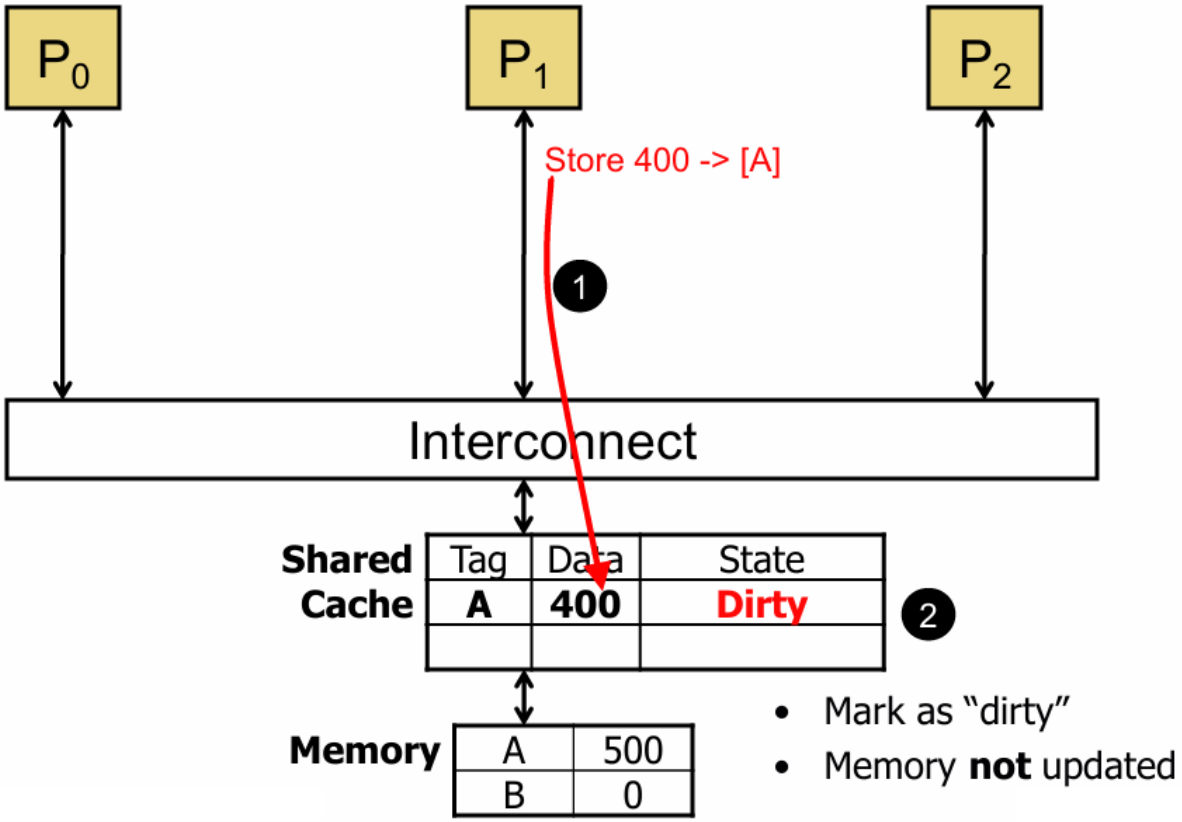
Shared Cache Implementation



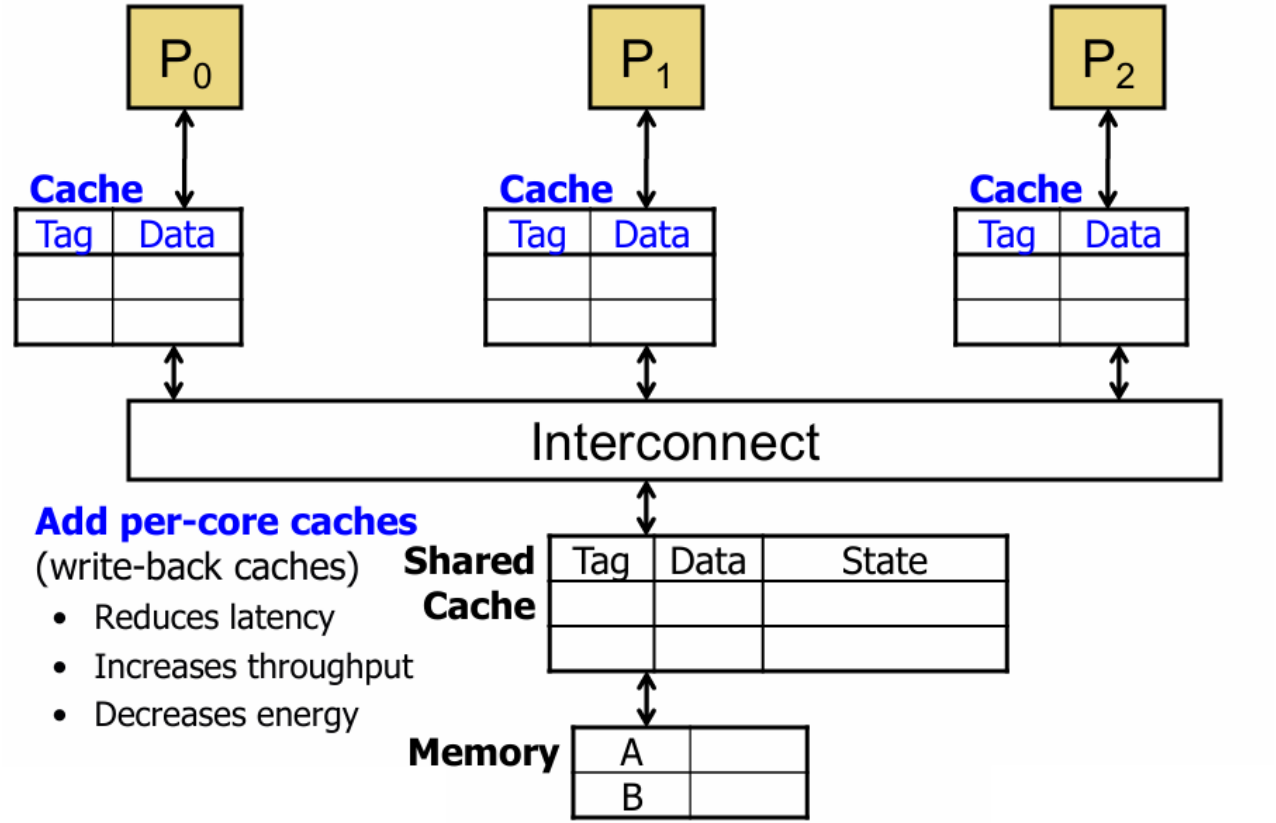
Shared Cache Implementation



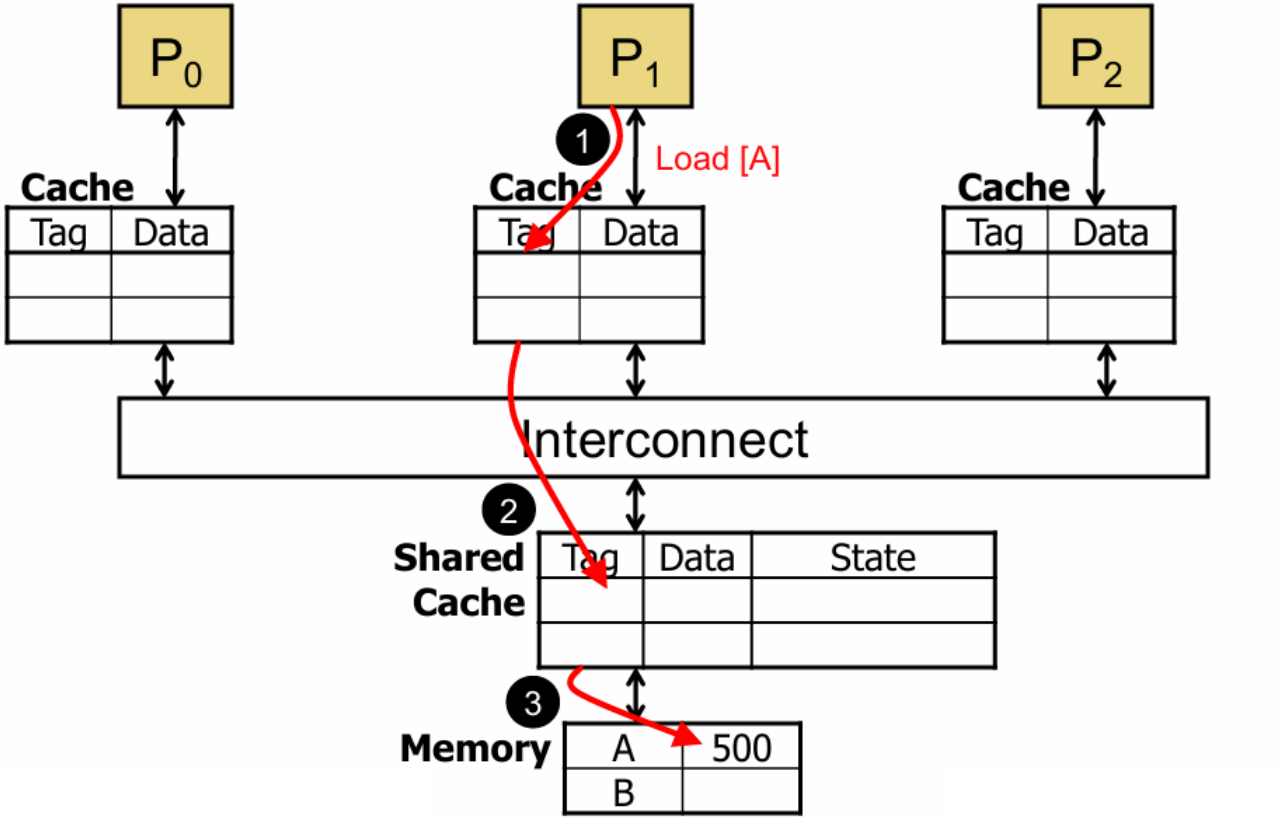
Shared Cache Implementation



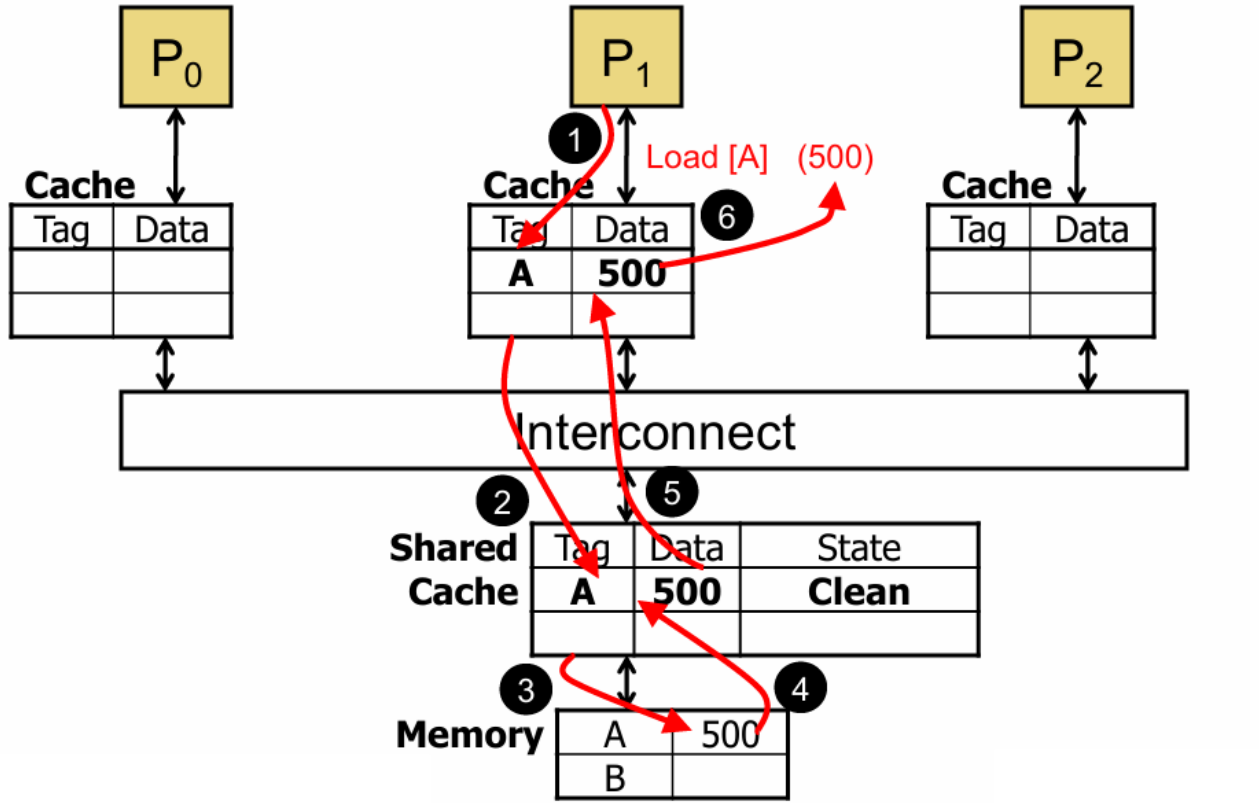
Adding Private Caches



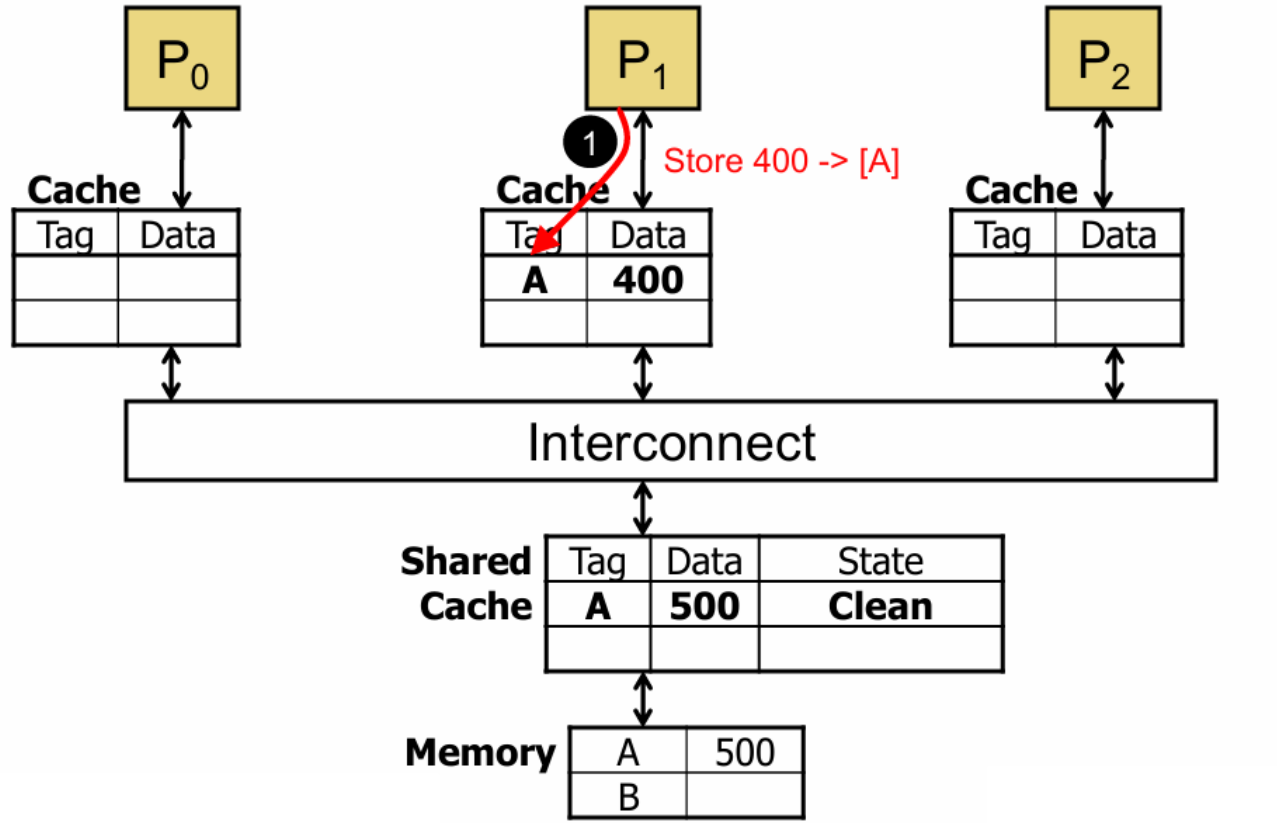
Adding Private Caches



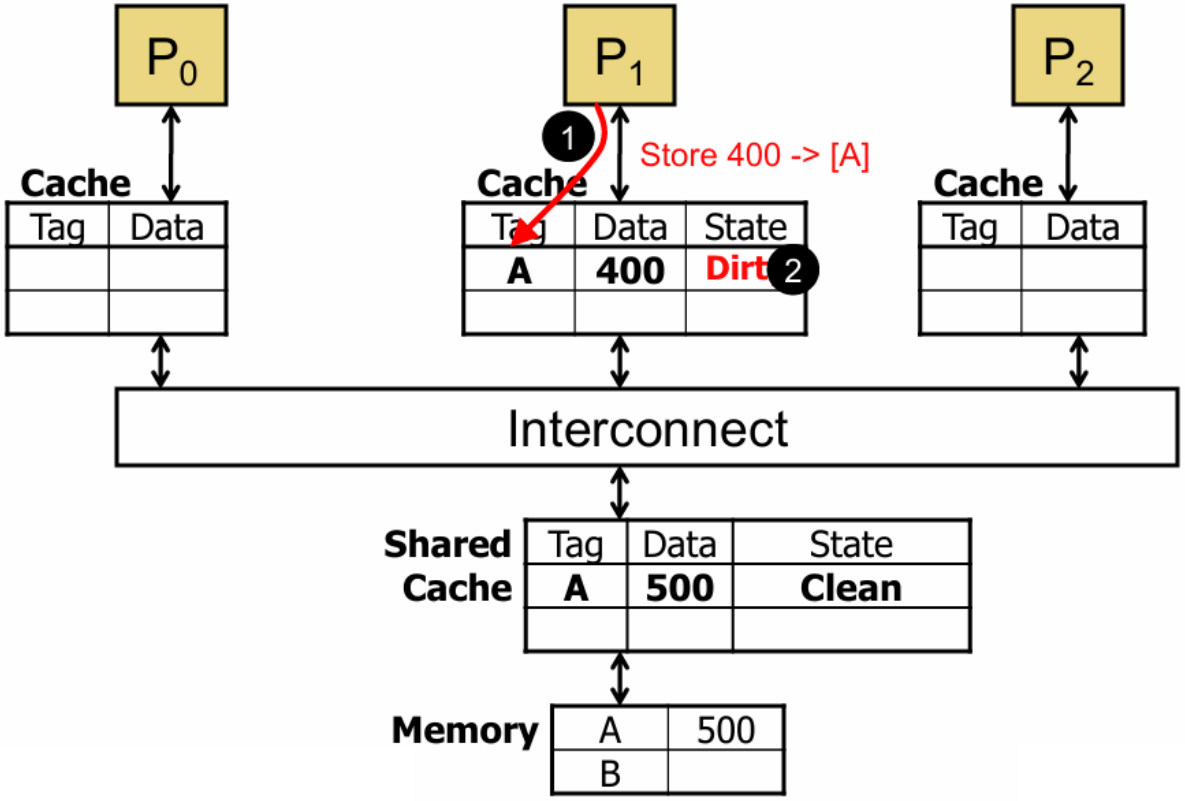
Adding Private Caches



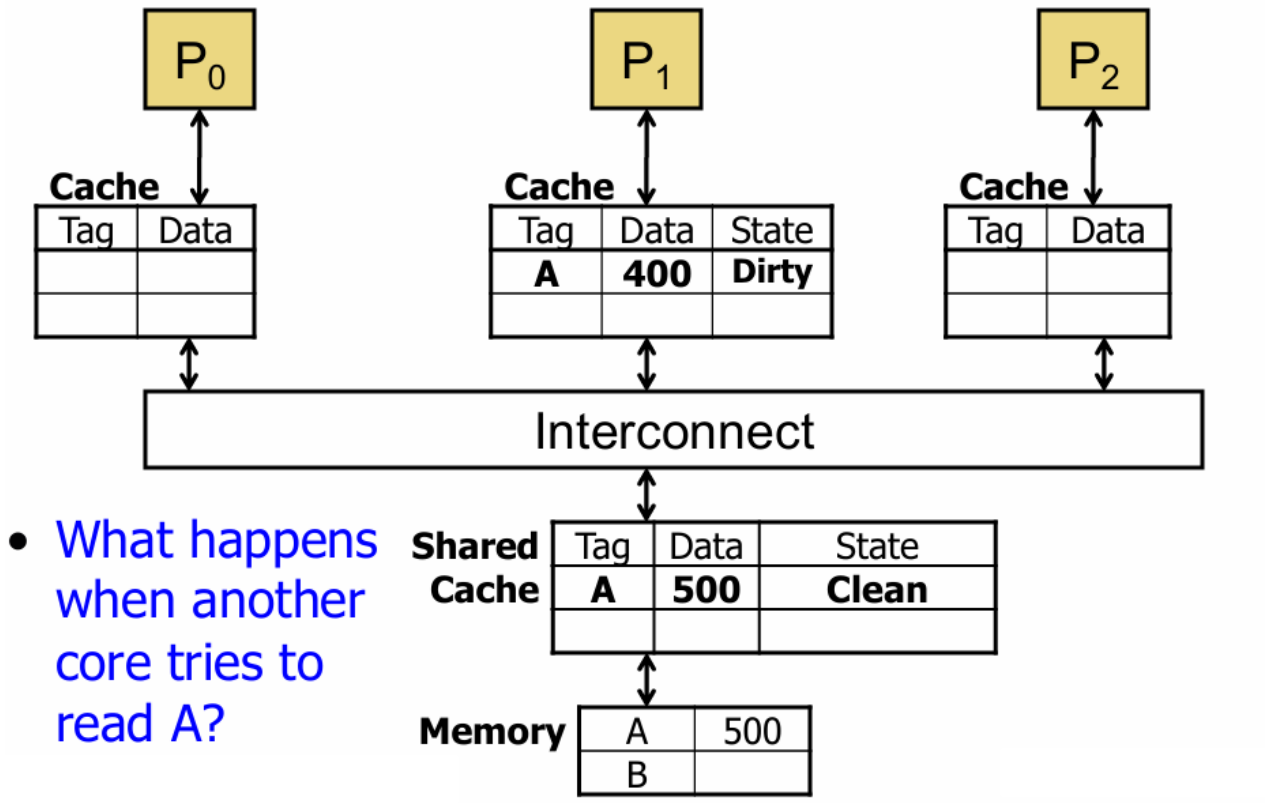
Adding Private Caches



Adding Private Caches

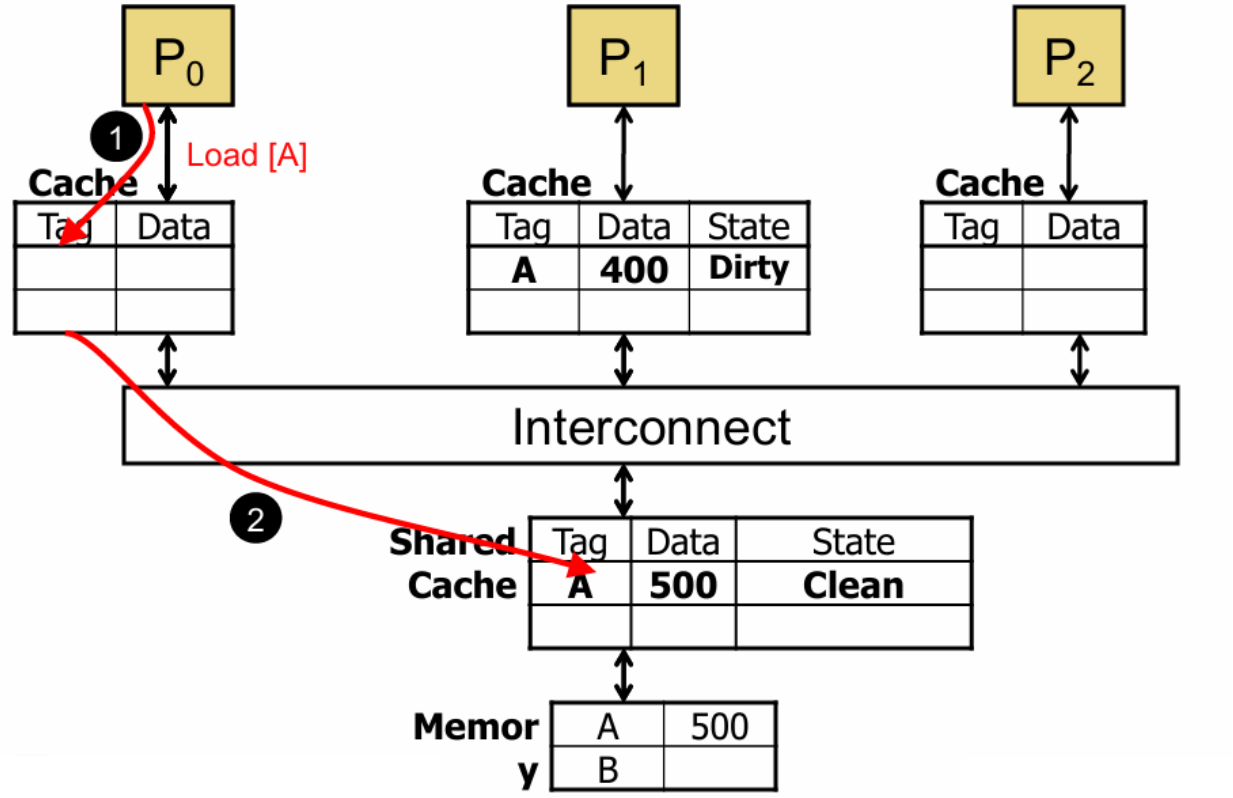


Private Cache Problem

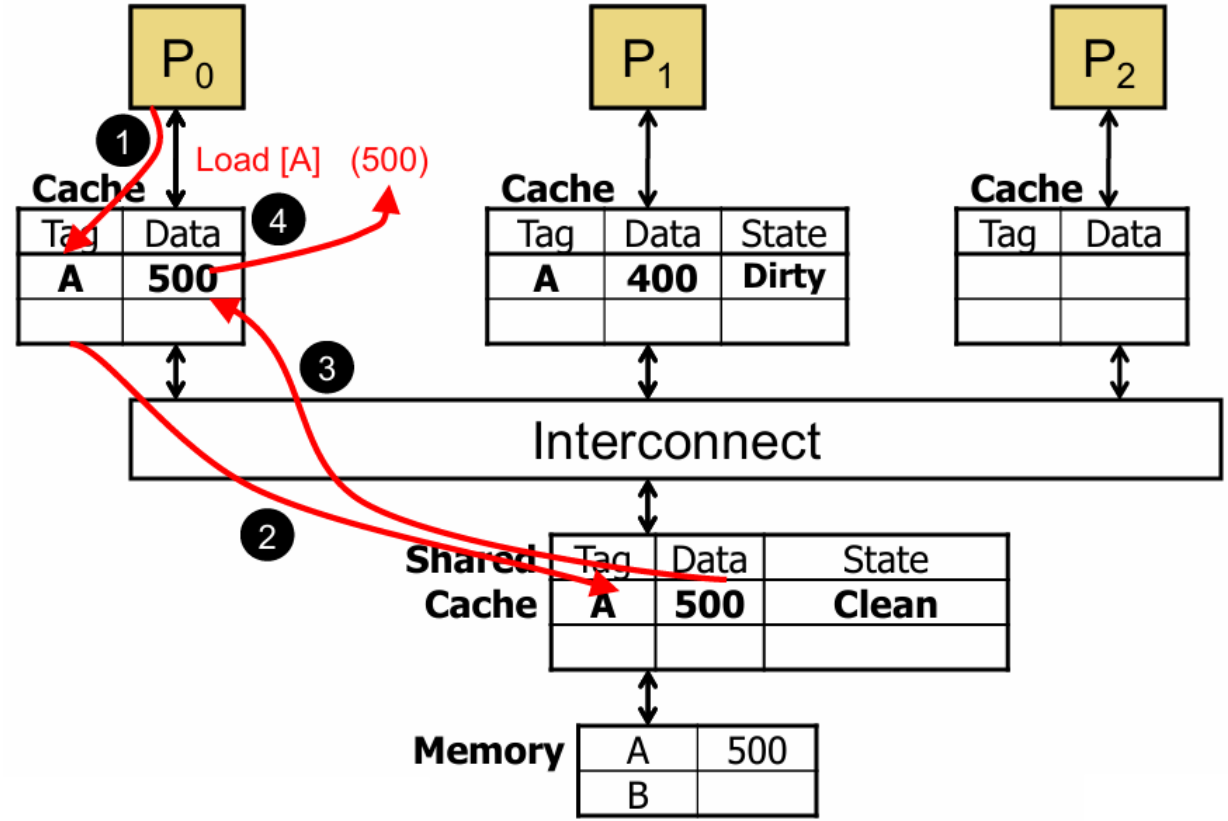


- What happens when another core tries to read A?

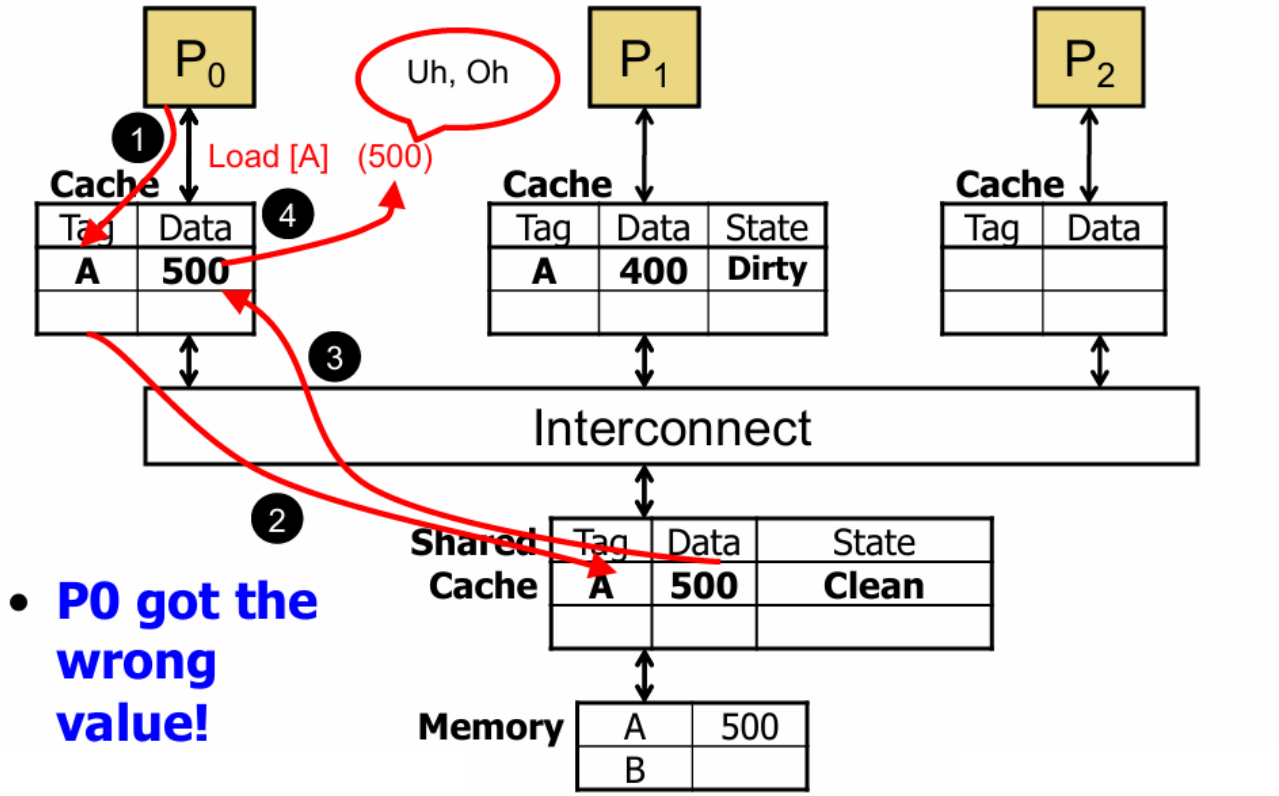
Private Cache Problem



Private Cache Problem

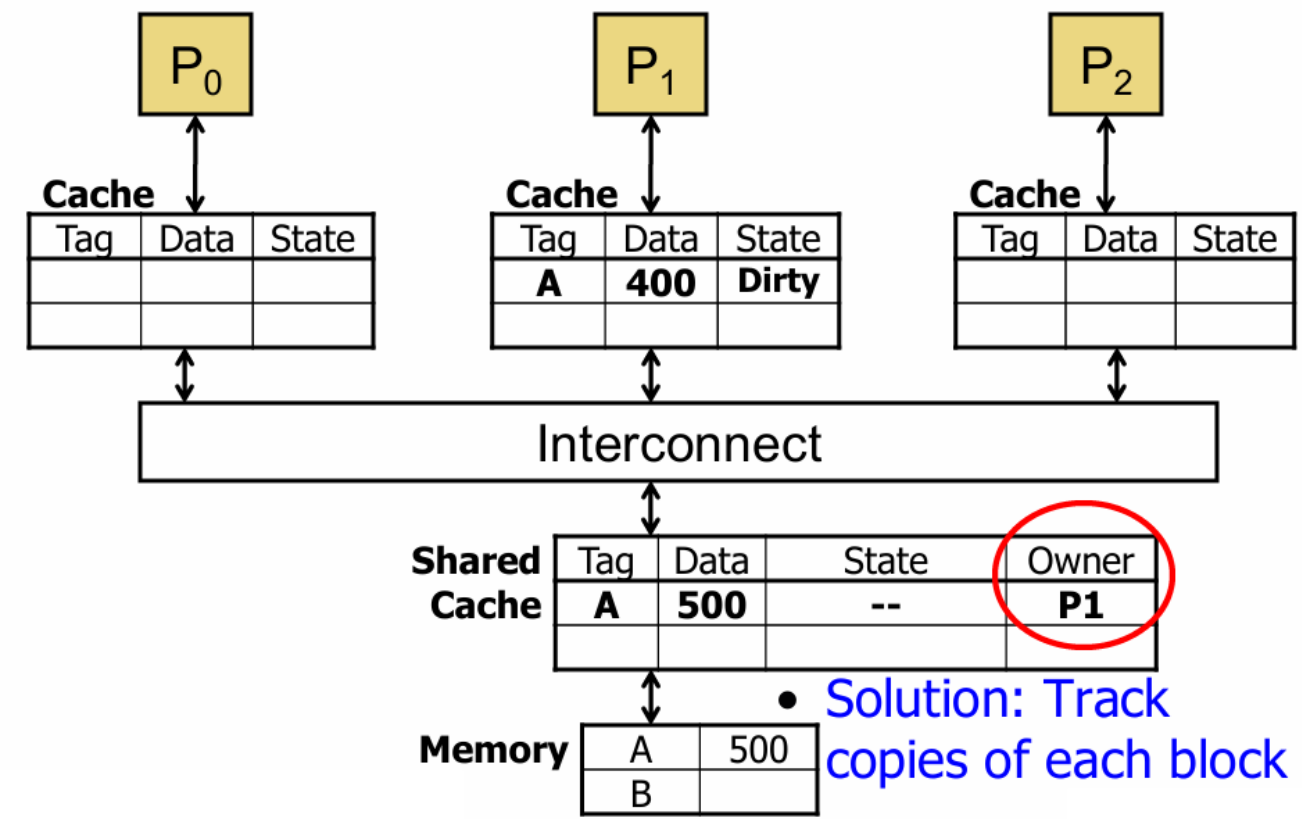


Private Cache Problem

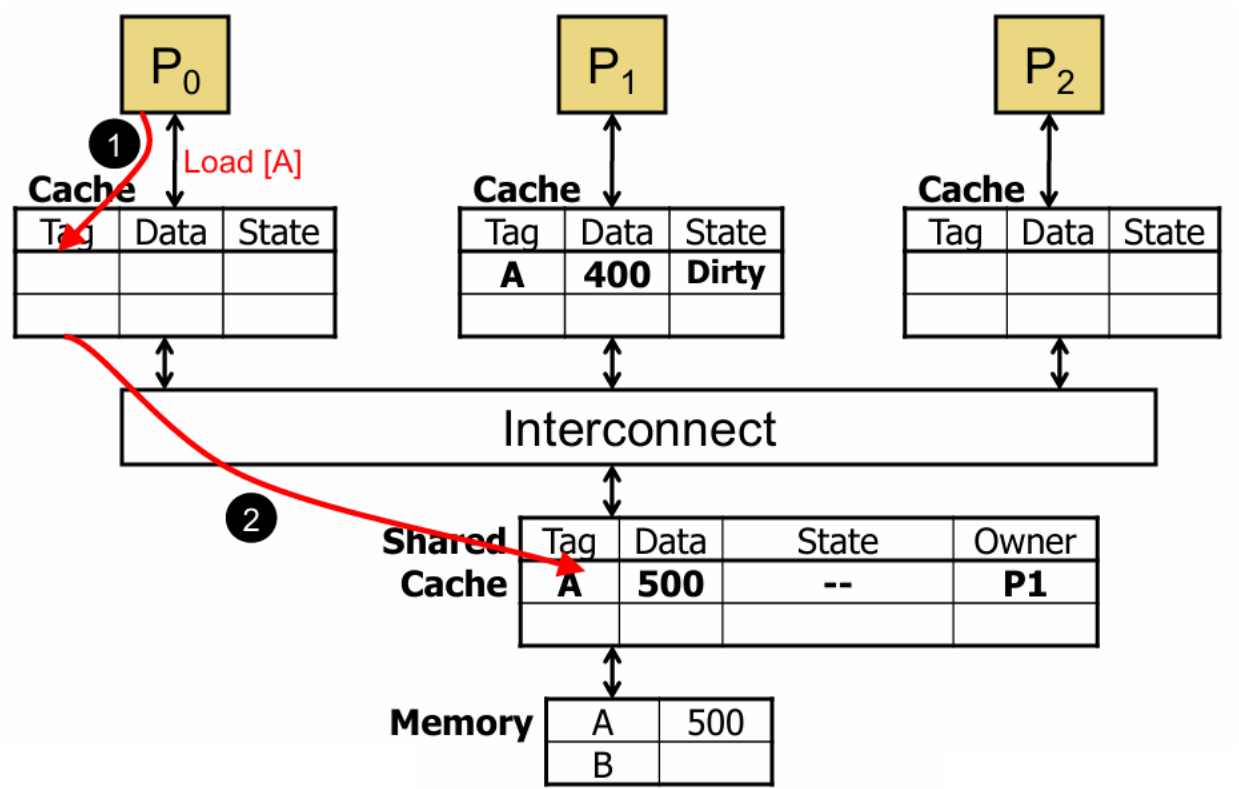


- **P0 got the wrong value!**

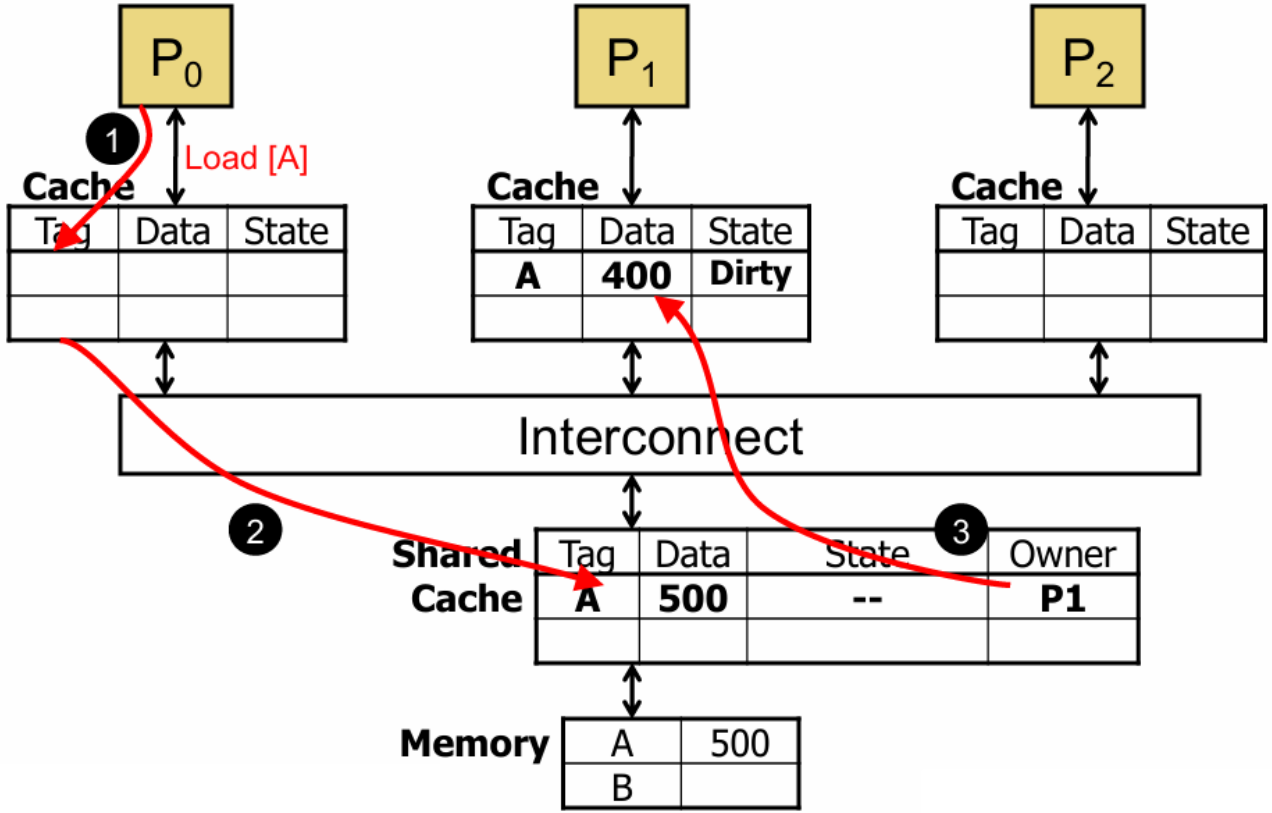
Fix Problem by Tracking Sharers



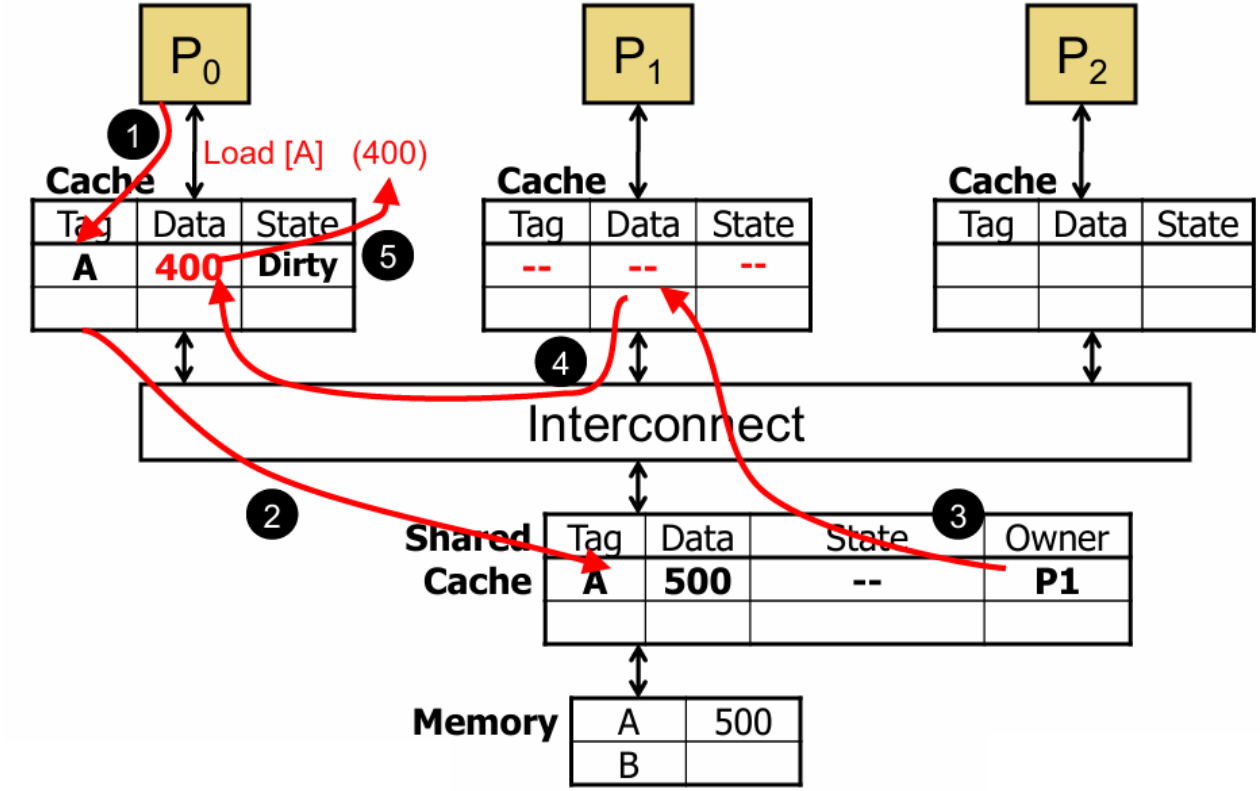
Use Tracking info to Invalidate



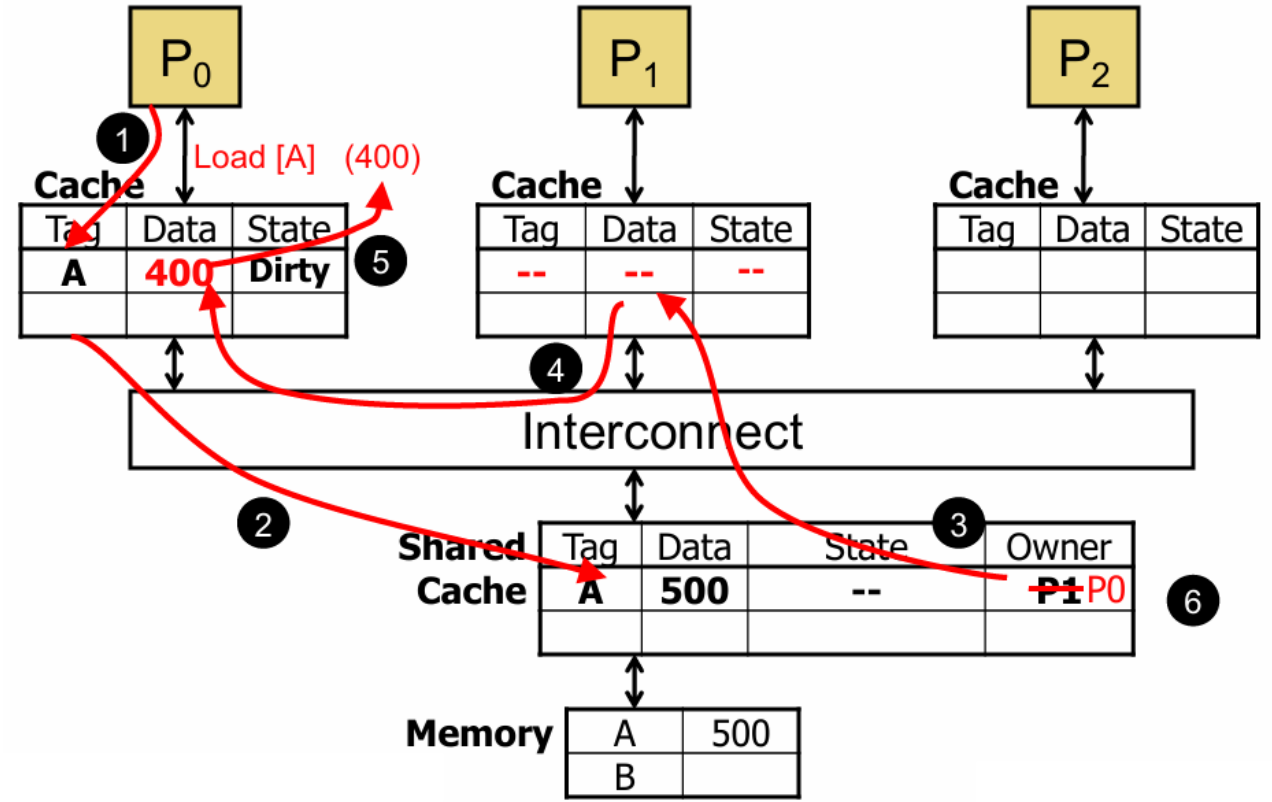
Use Tracking info to Invalidate



Use Tracking info to Invalidate



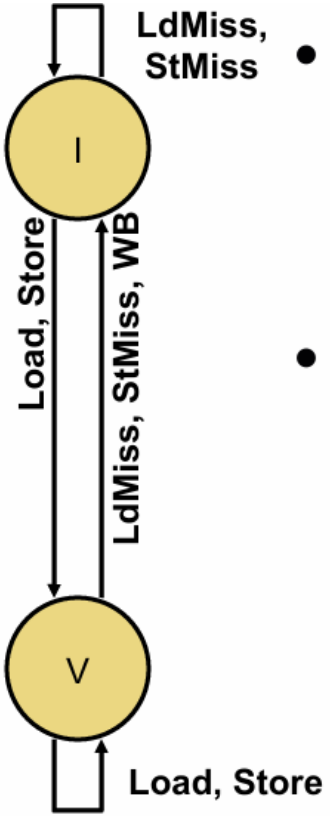
Use Tracking info to Invalidate



Valid/Invalid Cache Coherence

- To enforce the shared memory invariant…
  - *Loads read the value written by the most recent store*
- …we enforce the invariant
  - **At most one valid copy of a block**
  - Simplest form is a **two-state Valid/Invalid protocol**
  - If a core wants a copy, must find and invalidate it
- On a cache miss, how is the valid copy found?
  - **Snooping**: broadcast to all, whoever has it responds
  - **Directory**: track sharers explicitly
- **Problem**: multiple read-only copies not allowed
  - Makes read-only data, code, etc. slow

VI (MI) Coherence Protocol



- **VI (valid-invalid) protocol**
  - Two states (per block in cache)
    - **V (valid)**: have block
    - **I (invalid)**: don't have block

    +Can implement with valid bit
- Protocol diagram (at left)
  - Summary
    - If anyone wants to read/write block
    - Give it up: transition to **I** state
    - Write-back if your own copy is dirty

VI Protocol State Transition Table

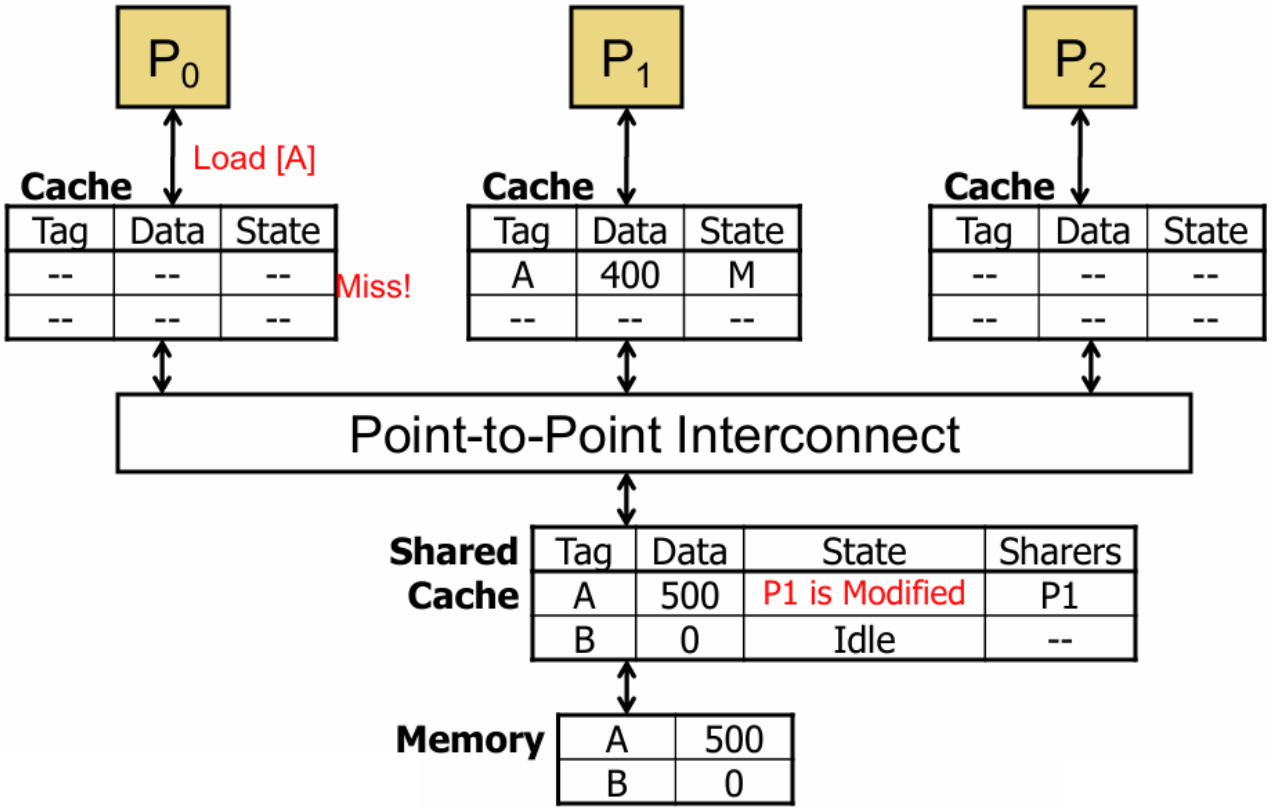
State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Load Miss ➔ V	Store Miss ➔ V	---	---
Valid (V)	Hit	Hit	Send Data ➔ I	Send Data ➔ I

- Rows are "states"
  - I vs V
- Columns are "events"
  - Writeback events not shown
- Memory controller not shown
  - **Memory sends data when no processor responds**

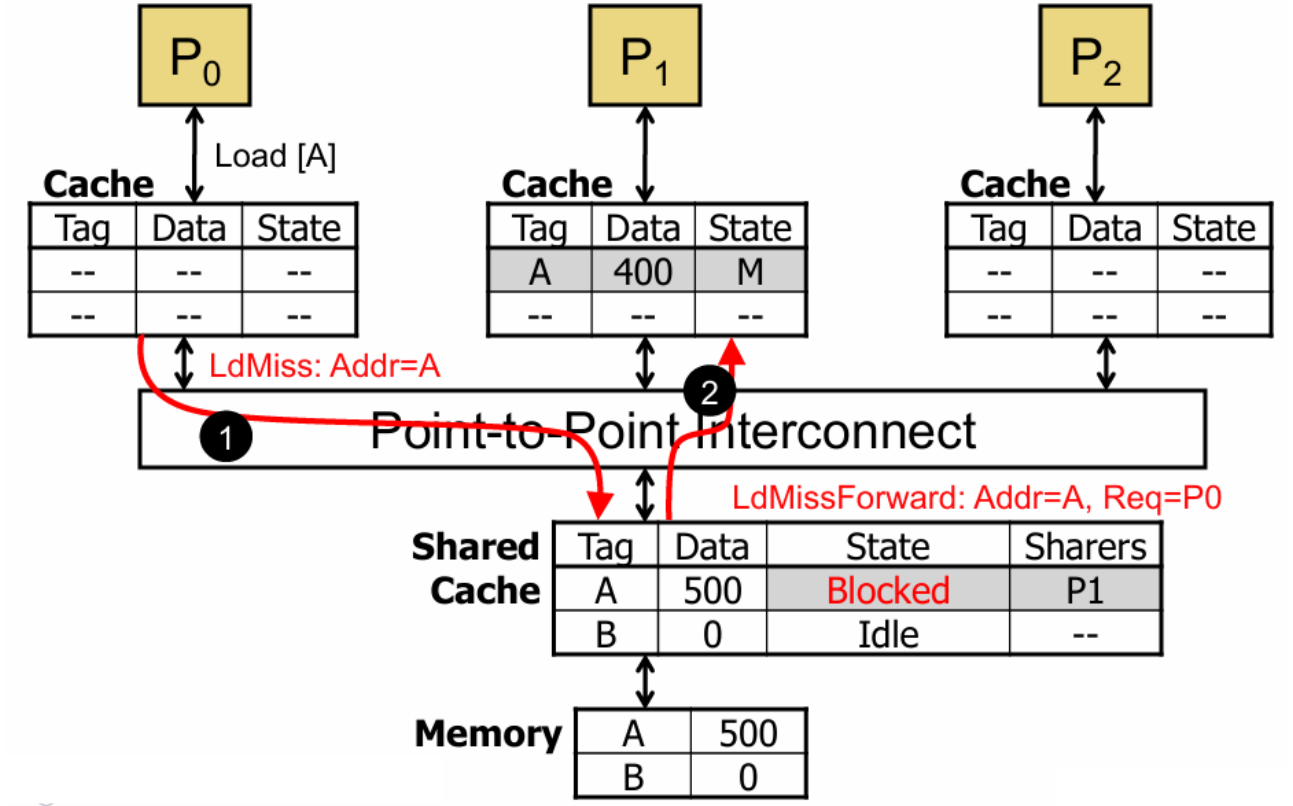
MSI Cache Coherence Protocol

- Add an extra state to track read-only lines
  - **Multiple read-only copies, or a single read/write copy**
  - **Modified (M): read/write permission**
  - **Shared (S): read-only permission**
  - **Invalid (I): no permission**
- Also track a **Sharer bit vector** in shared cache
  - One bit per core; tracks all shared copies of a block
  - When a write occurs, must invalidate all readers
- Allows for many readers…
  - …while still enforcing shared memory invariant (*Loads read the value written by the most recent store*)

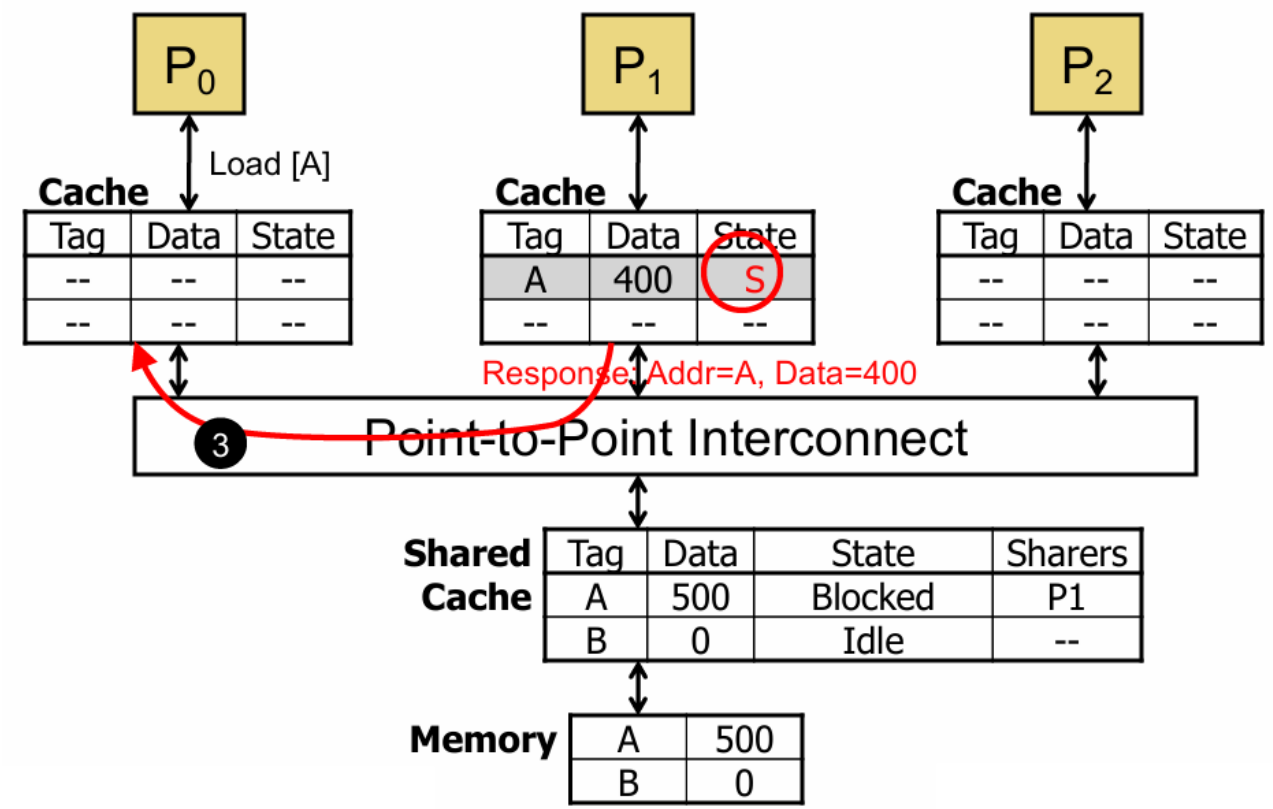
MSI Coherence Example: Step #1



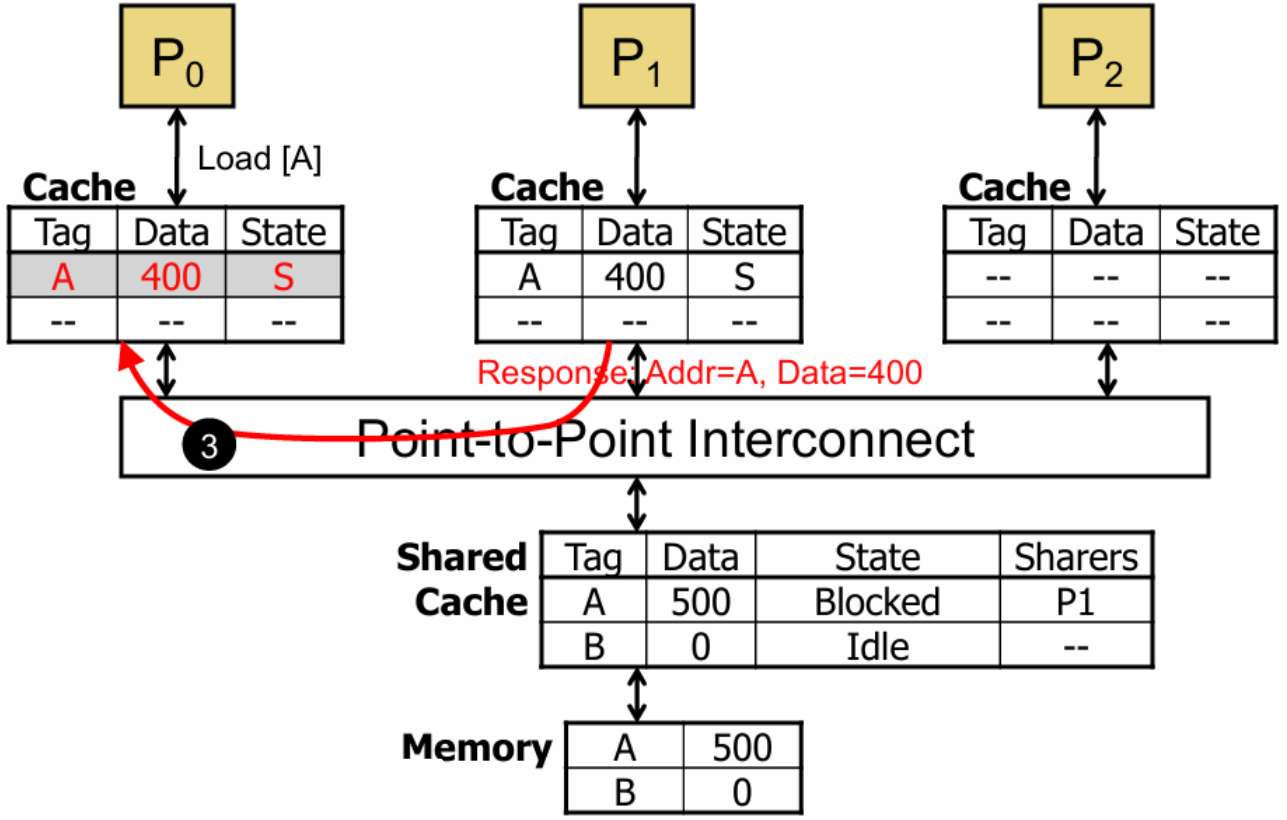
MSI Coherence Example: Step #2



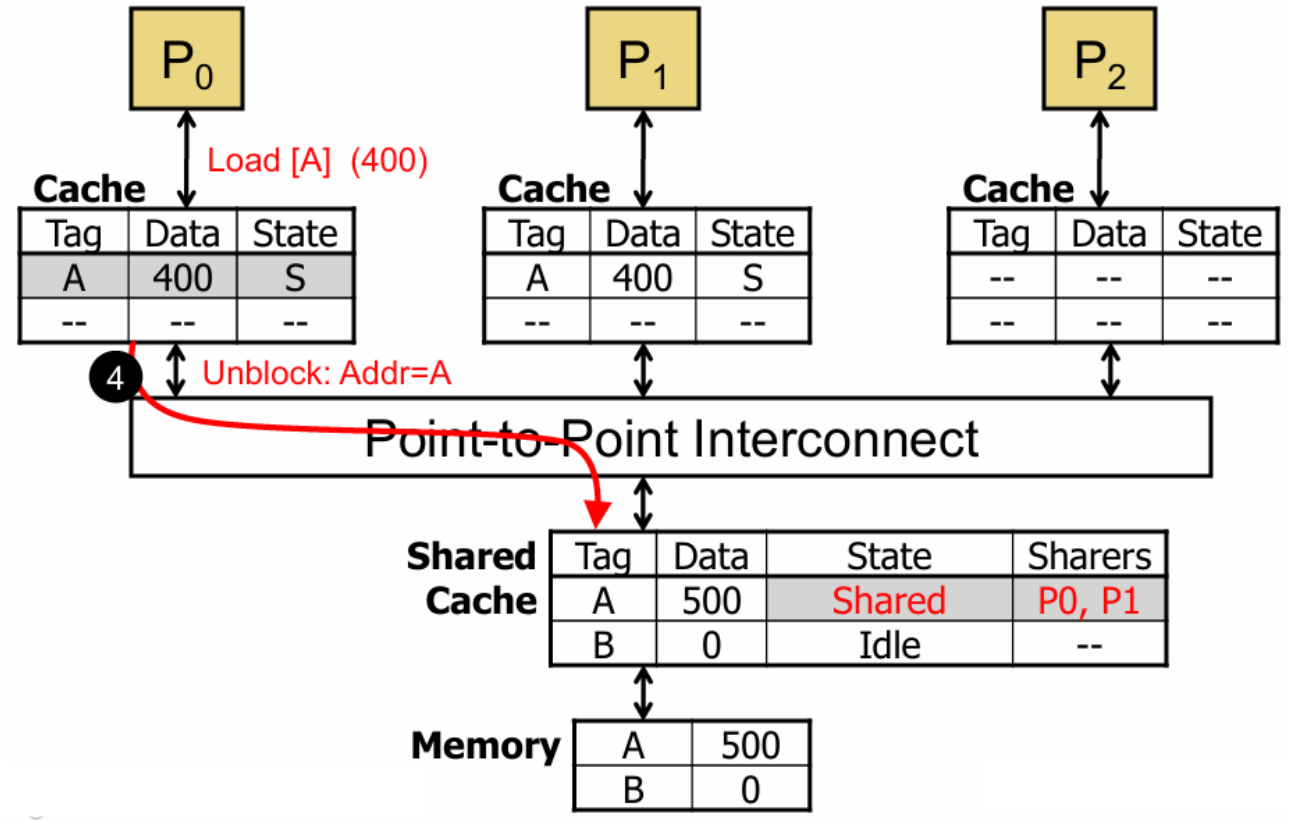
MSI Coherence Example: Step #3



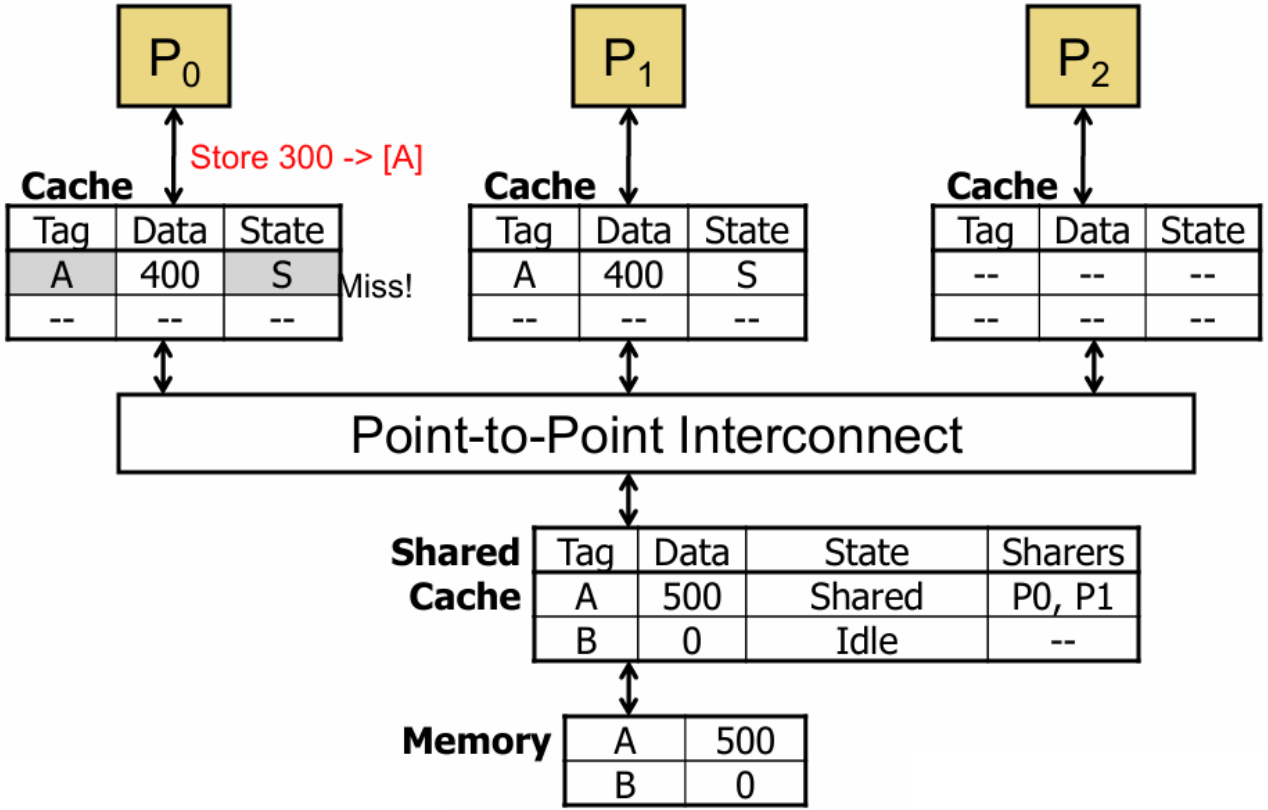
MSI Coherence Example: Step #4



MSI Coherence Example: Step #5



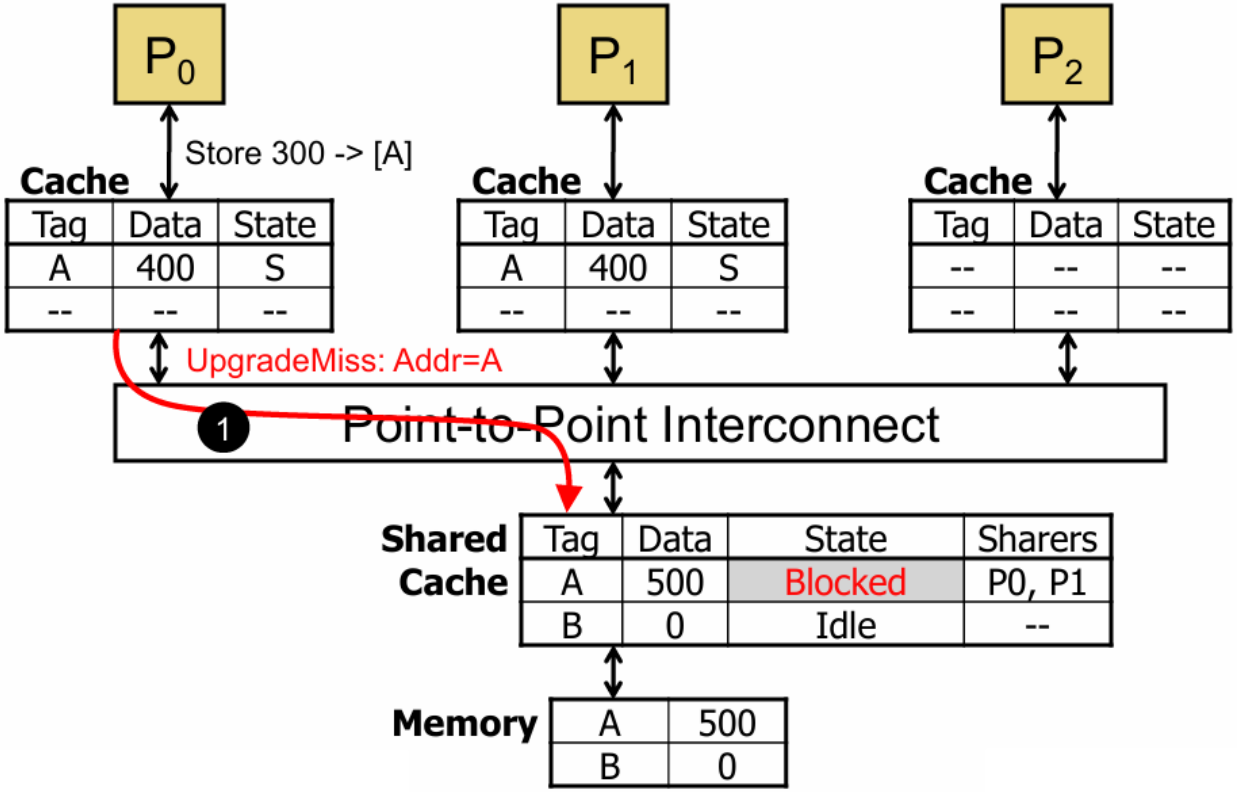
MSI Coherence Example: Step #6



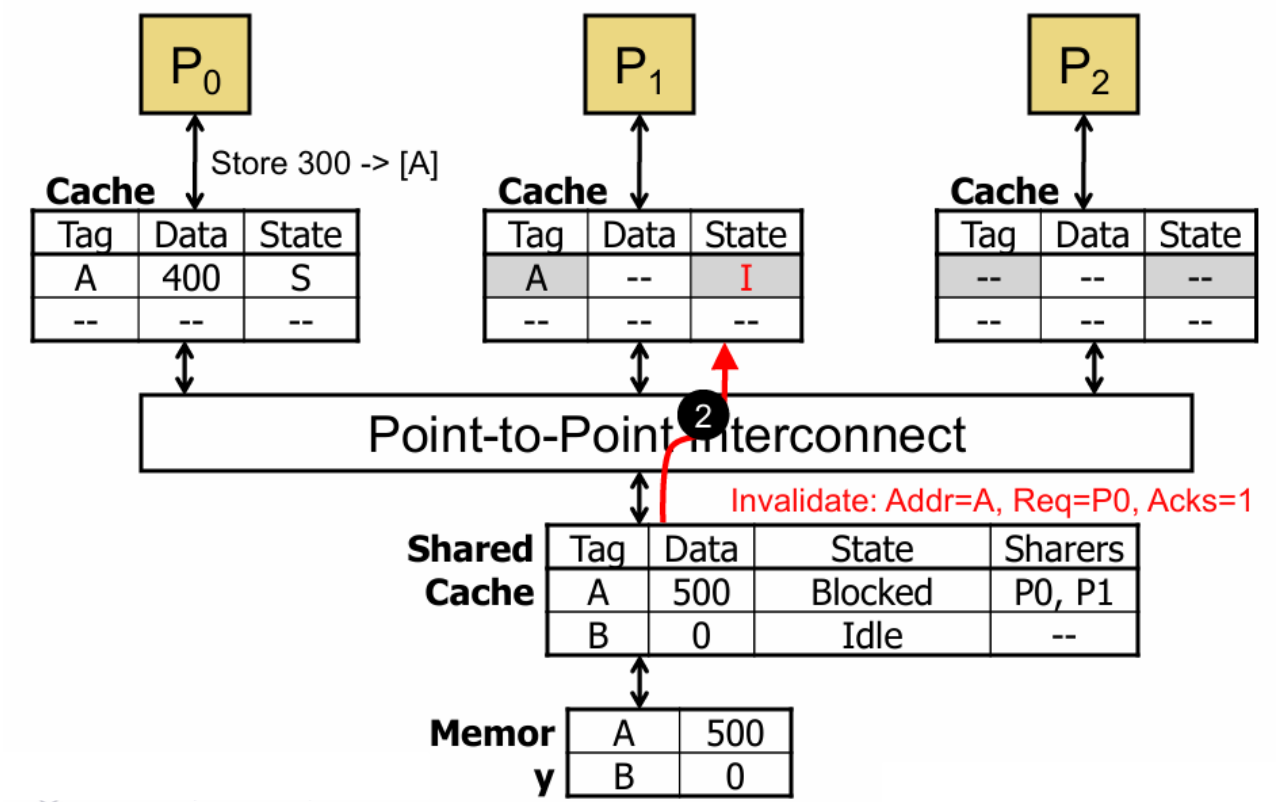
Classifying Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
  - **Capacity**: miss caused because cache is too small
    - Would miss even in fully associative cache
  - **Conflict**: miss caused because cache associativity is too low
    - All misses that aren't compulsory or capacity
  - **COHERENCE: miss due to external invalidations**

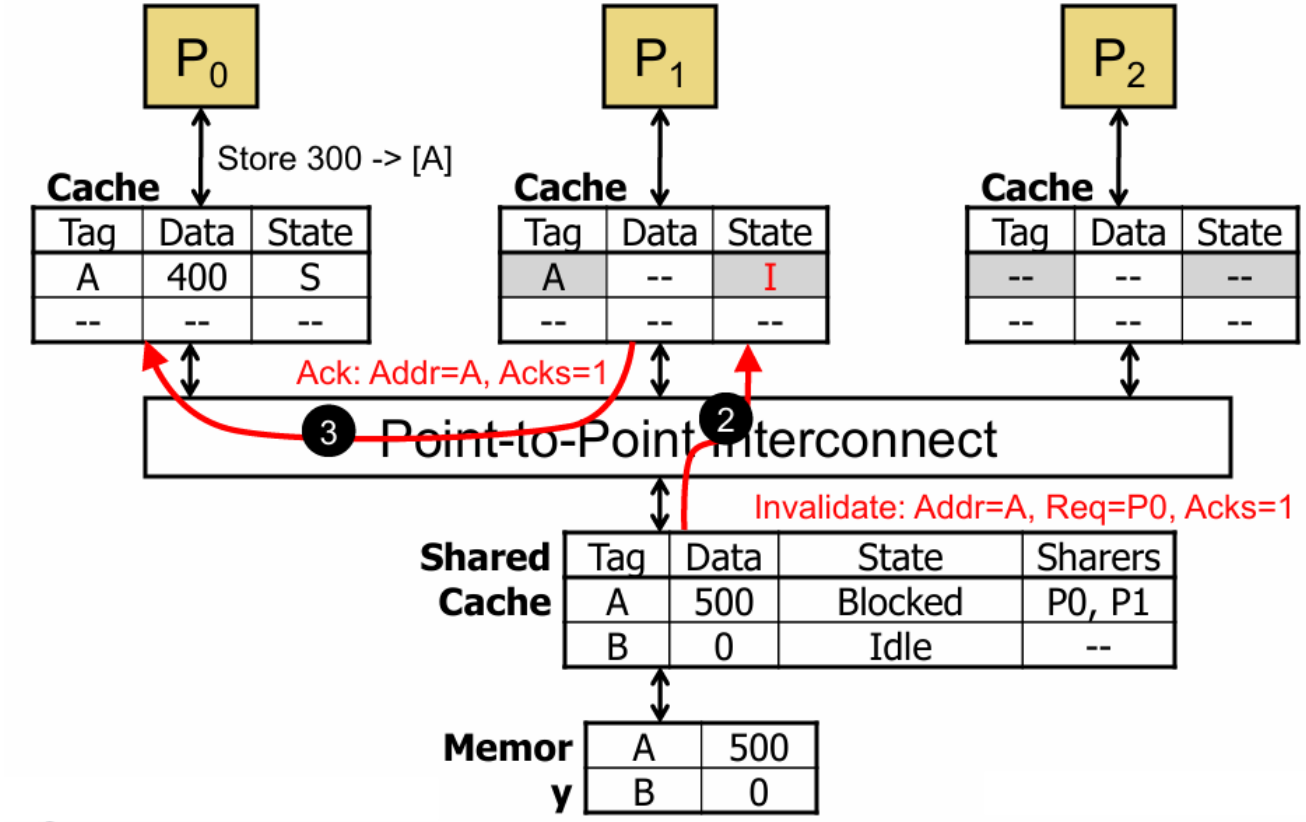
MSI Coherence Example: Step #7



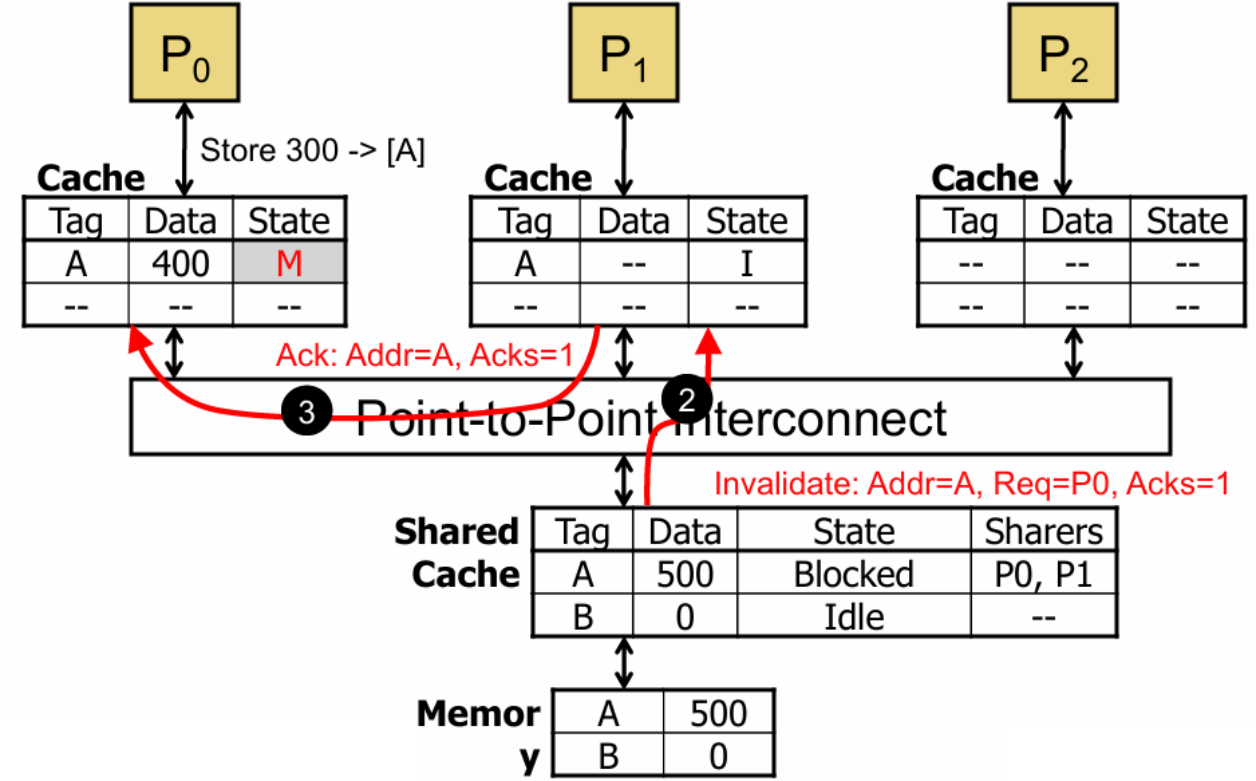
MSI Coherence Example: Step #8



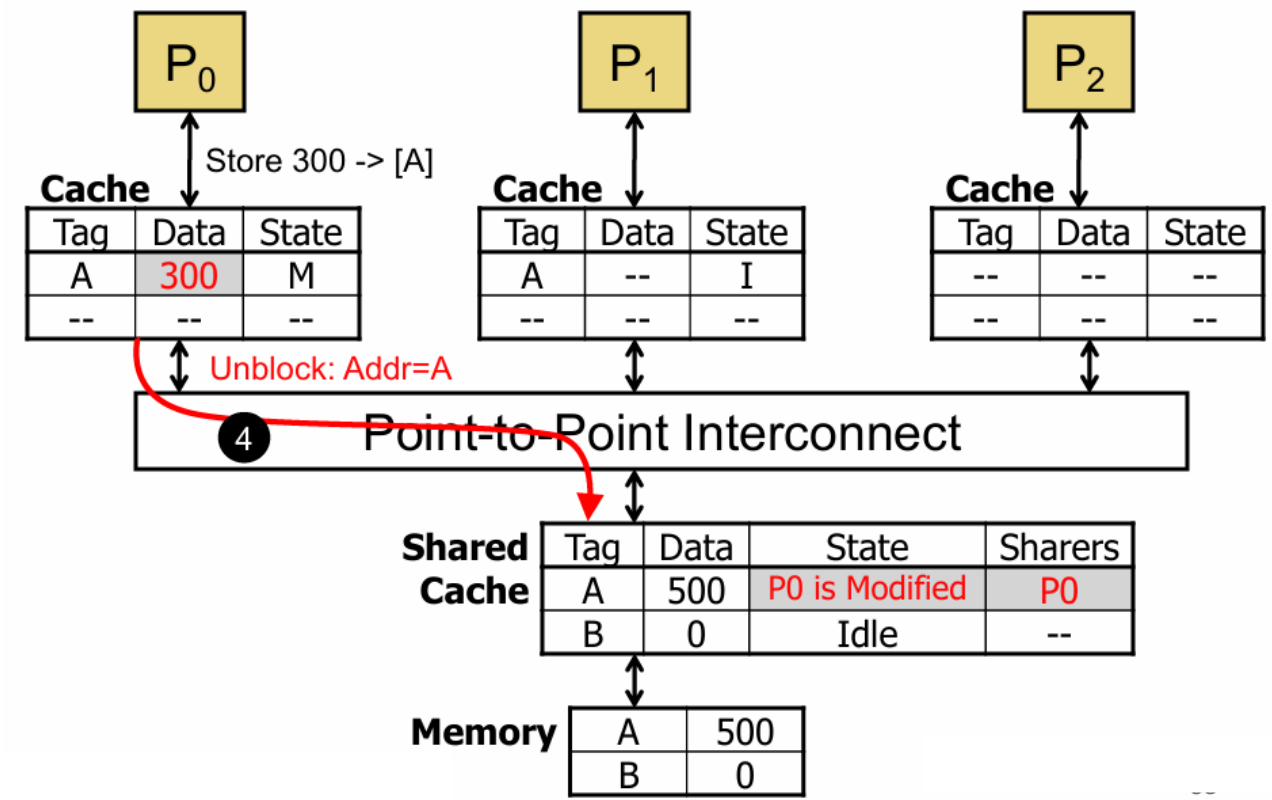
MSI Coherence Example: Step #9



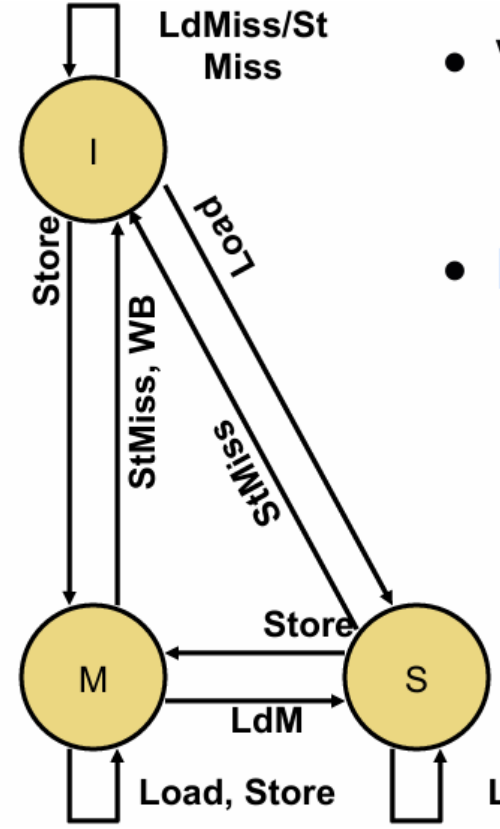
MSI Coherence Example: Step #10



MSI Coherence Example: Step #11



VI-> MSI



- VI protocol is inefficient
  - Only one cached copy allowed
  - Disallows multiple read-only copies
- **MSI (modified-shared-invalid)**
  - Splits V state into two states
    - **M (modified)**: local dirty copy
    - **S (shared)**: local clean copy
  - Allows **either**
    - Multiple read-only copies (S-state), or
    - Single read/write copy (M-state)

MSI Protocol State Transition Table

State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Load Miss → S	Store Miss → M	---	---
Shared (S)	Hit	Upgrade Miss → M	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

Cache Coherence and Cache Misses

- Coherence introduces two new kinds of misses
  - **Upgrade miss:** stores to read-only blocks
    - Delay to acquire write permission to read-only block
  - **Coherence miss**
    - Miss to a block evicted by another processor's requests
- Making the cache larger…
  - Doesn't reduce these types of misses
  - So, as cache grows large, these sorts of misses dominate
- **False sharing**
  - Two or more processors sharing parts of the same block
  - But *not* the same bytes within that block (no actual sharing)
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult

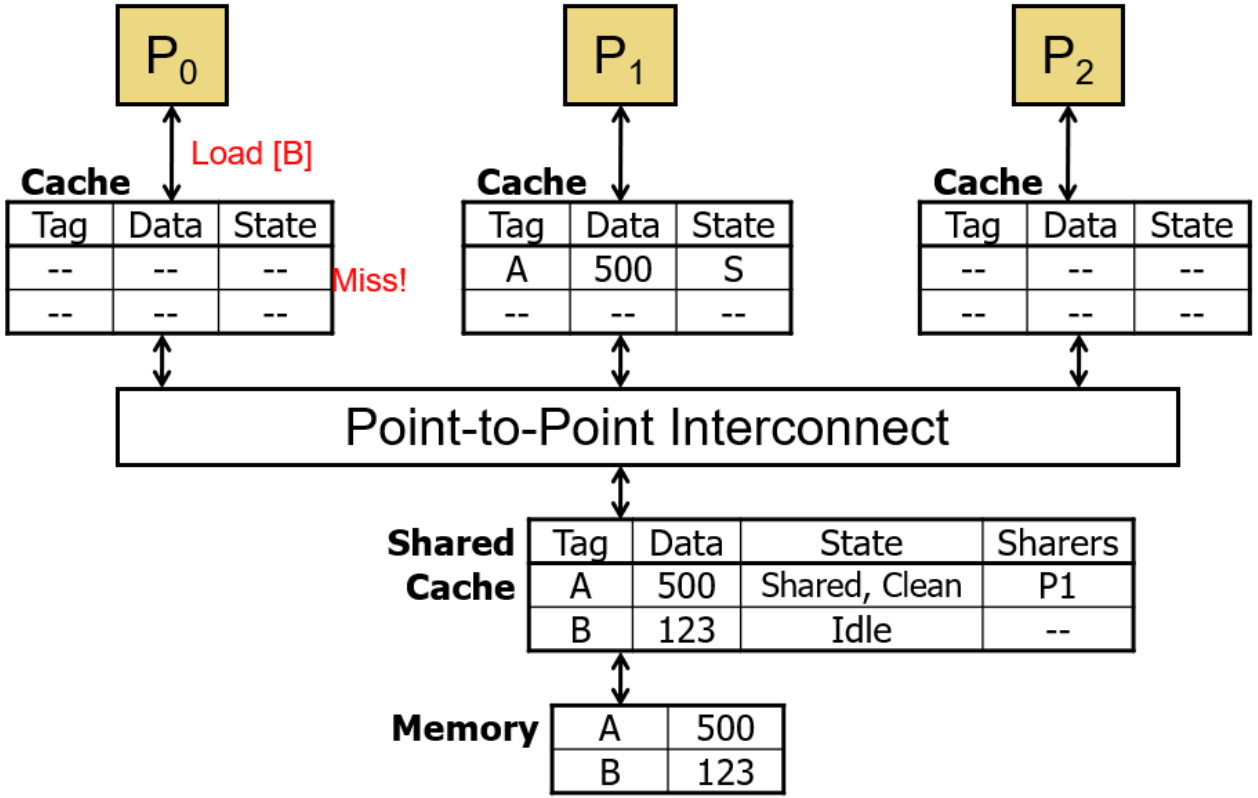
MESI Cache Coherence

- Consider load followed by store by same core
  - **two misses**: load miss plus an upgrade miss…
  - … even if the block is never shared!
  - Some programs have lots of private data
    - E.g., single-threaded programs
    - Potentially **doubling** number of cache misses
- Solution:
  - Add an **Exclusive** state
  - "I have the **only** cached copy, and it's a **clean** copy"
    - Has read/write permissions
    - Just like Modified but clean instead of dirty

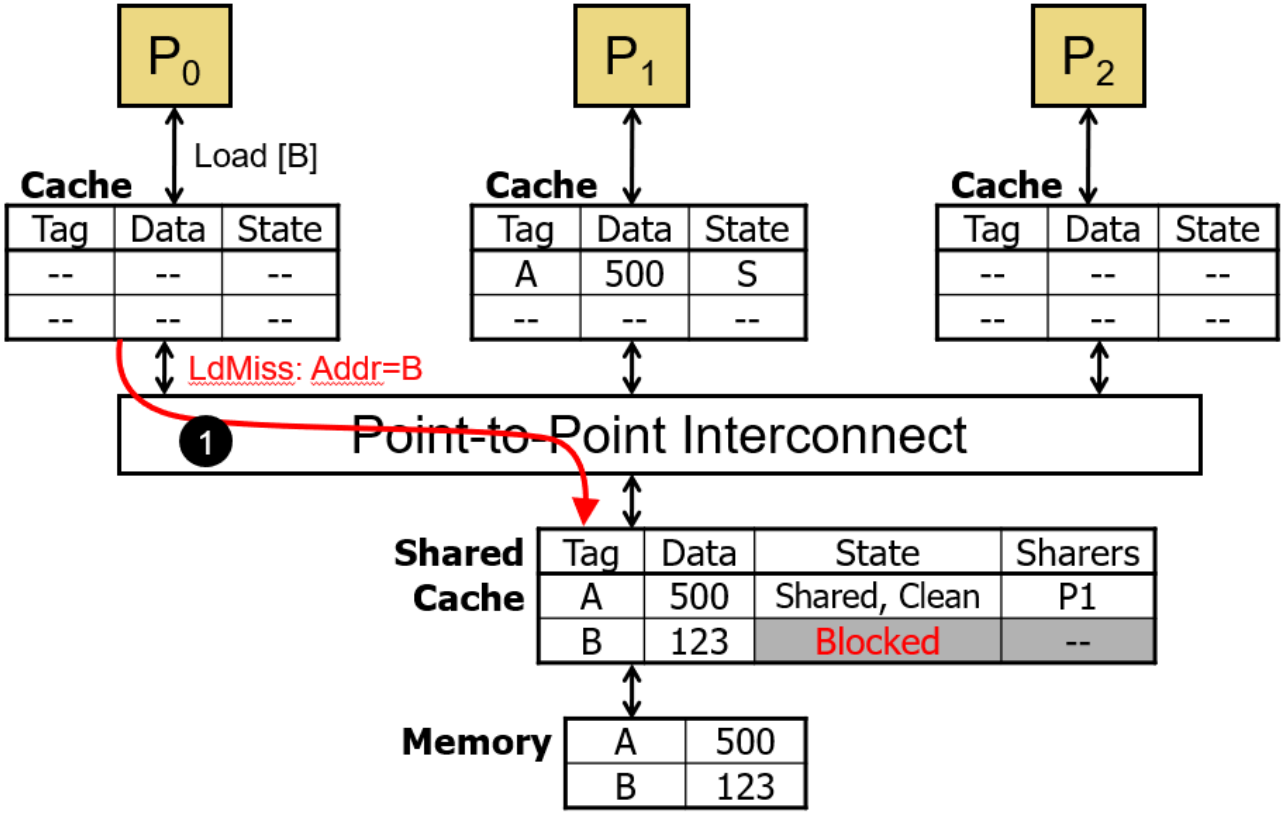
MESI Operations

- Goals:
  - Avoid “upgrade” misses for non-shared blocks
  - While not increasing eviction (aka writeback or replacement) traffic
- Two cases on a load miss to a block…
  - **Case #1**: … with no current sharers
    (that is, no sharers in the set of sharers)
    - Grant requester Exclusive copy with read/write permission
  - **Case #2**: … with other sharers
    - As before, grant just a Shared copy with read-only permission
- A store to a block in Exclusive changes it to Modified
  - **Instantaneously & silently** (no latency or traffic)
- On block eviction (aka writeback or replacement)…
  - If Modified, block is dirty, must be written back to next level
  - If Exclusive, writing back the data is not necessary

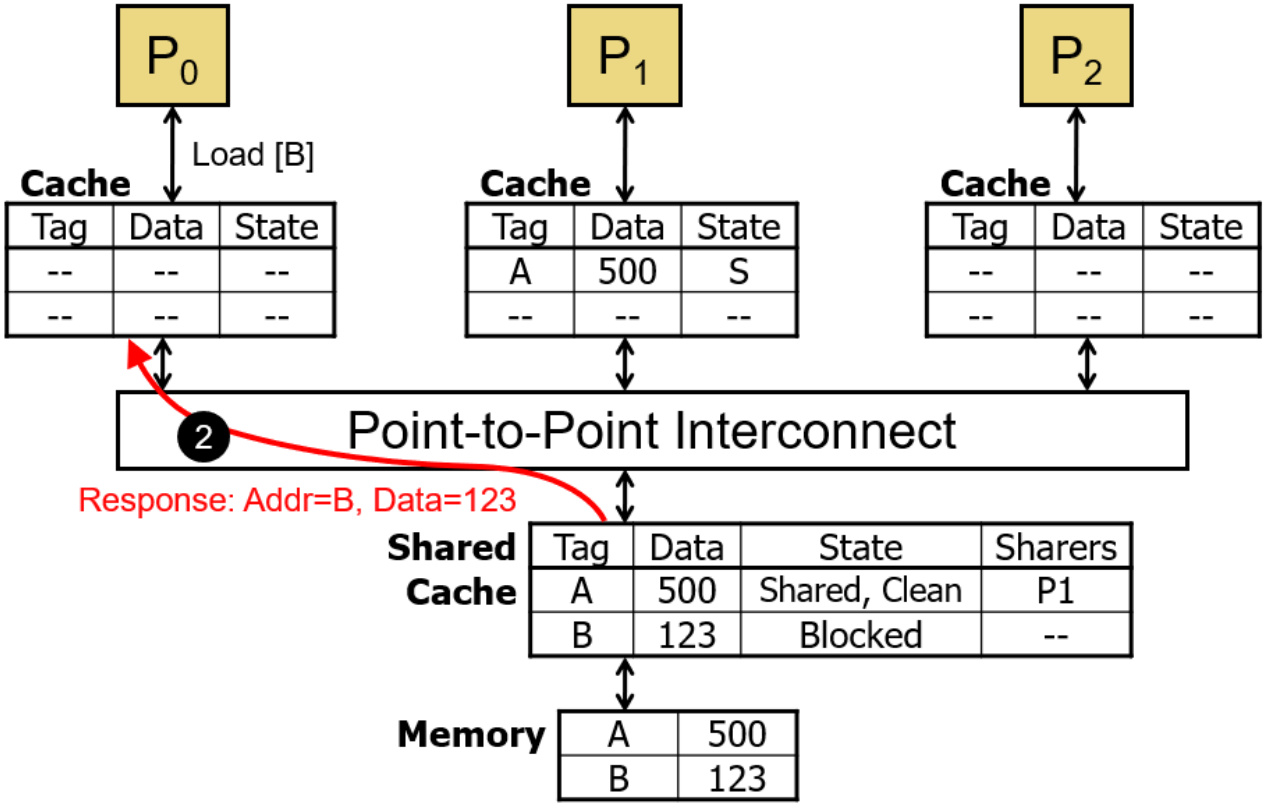
MESI Coherence Example: Step #1



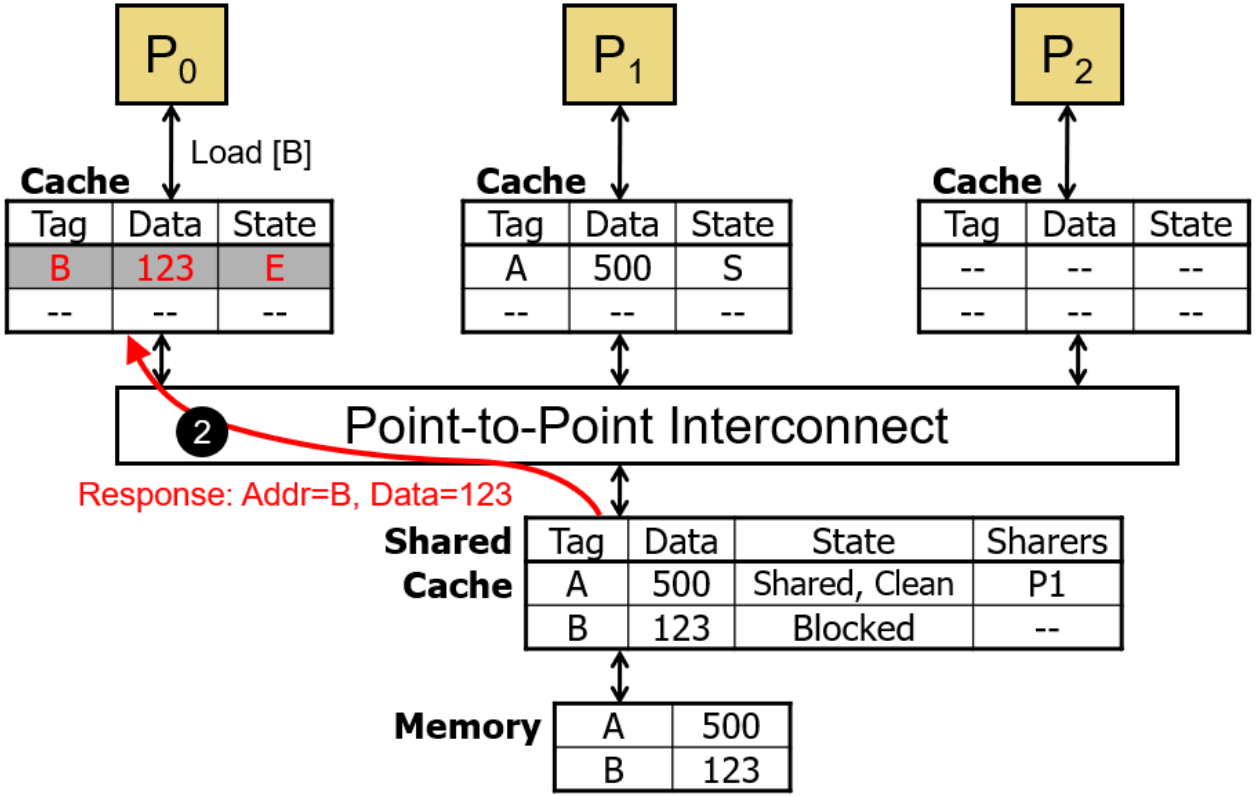
MESI Coherence Example: Step #2



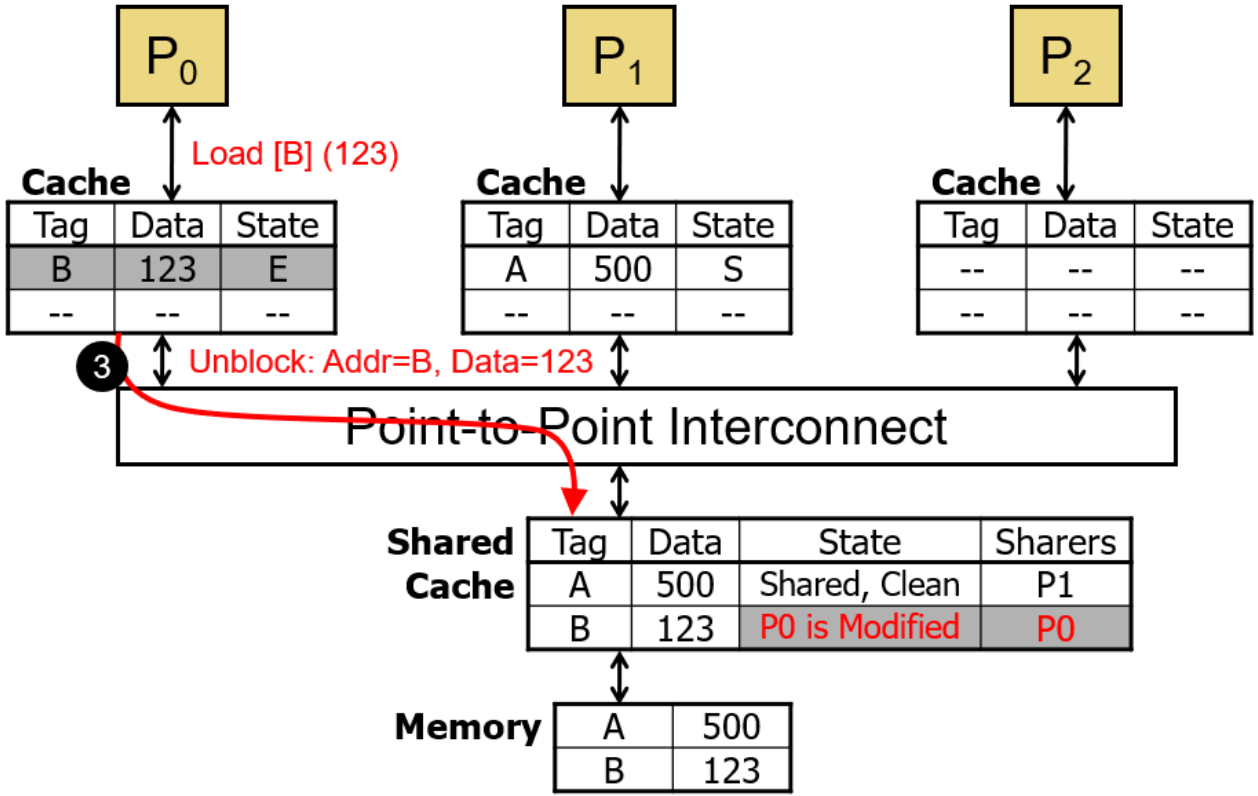
MESI Coherence Example: Step #3



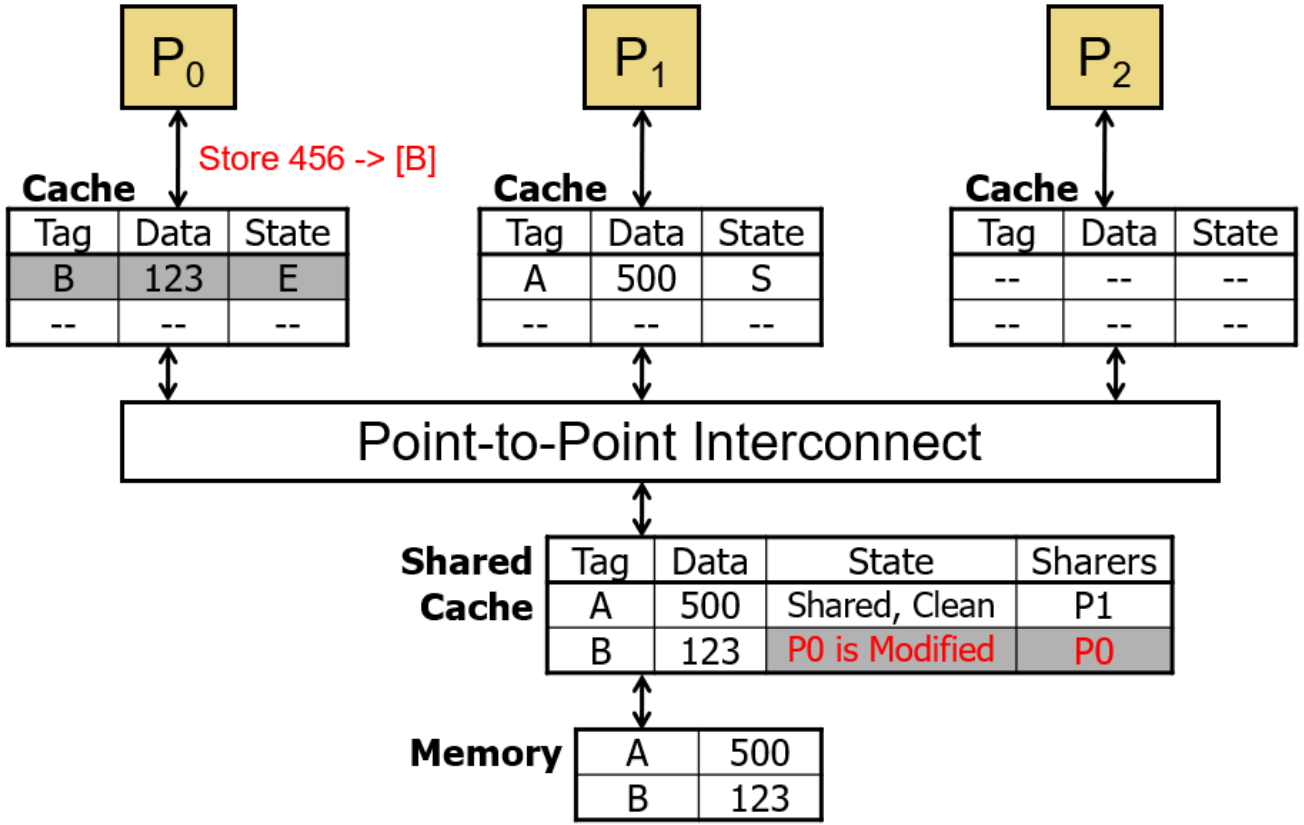
MESI Coherence Example: Step #4



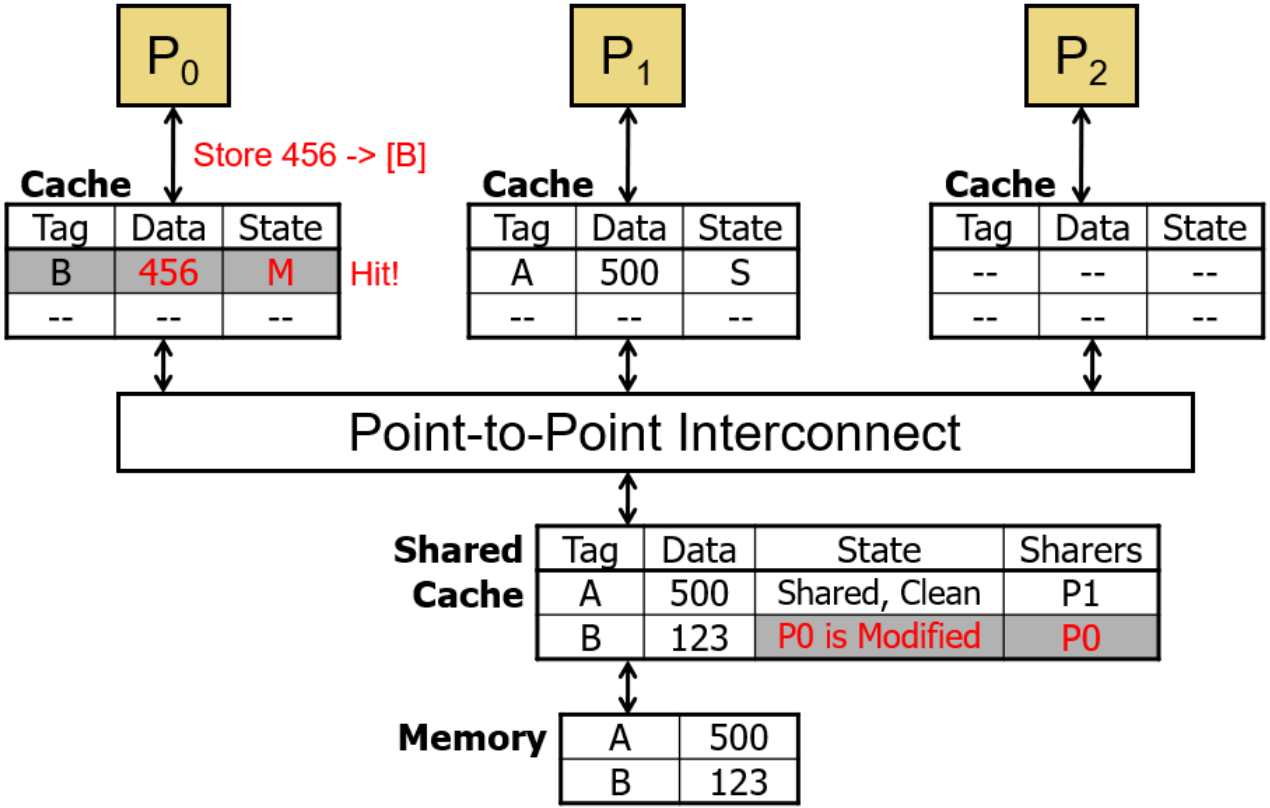
MESI Coherence Example: Step #5



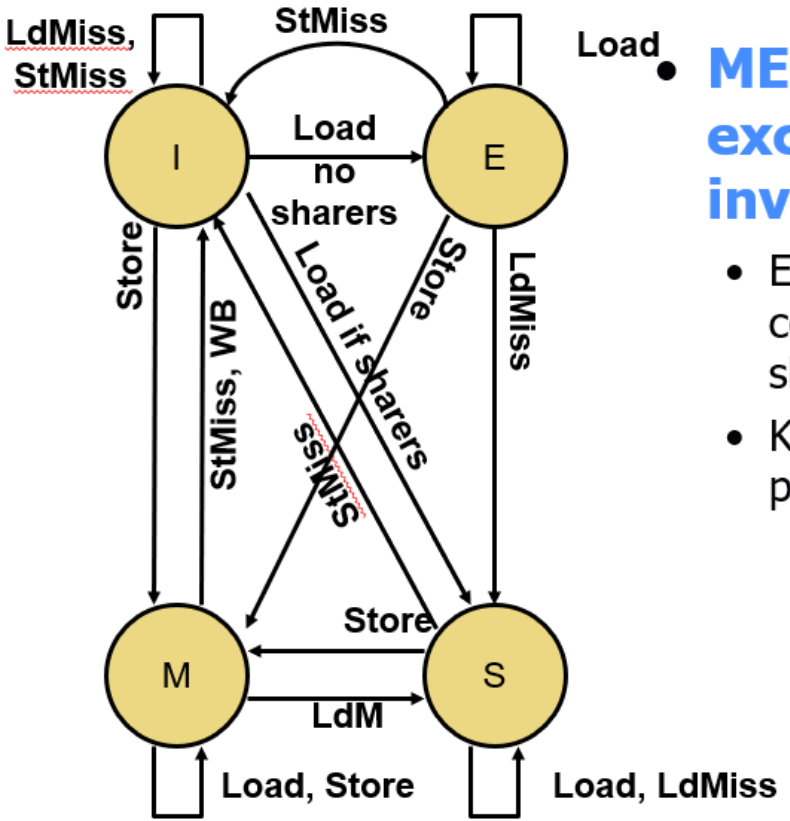
MESI Coherence Example: Step #6



MESI Coherence Example: Step #7



MSI -> MESI



- **MESI (modified-exclusive-shared-invalid)**
  - Eliminates the cost of coherence when there's no sharing
  - Keeps single-threaded programs fast on multicores

MESI State Transition Table

State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss ➔ **S or E**	Miss ➔ M	---	---
Shared (S)	Hit	Upg Miss ➔ M	Send Data ➔ S	Send Data ➔ I
Exclusive (E)	Hit	**Hit ➔ M**	Send Data ➔ S	Send Data ➔ I
Modified (M)	Hit	Hit	Send Data ➔ S	Send Data ➔ I

- Load misses lead to "E" if no other processors is caching the block

How to Compute SAXPY Quickly?

- Performing the **same** operations on **many** data items
  - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

```
L1: ldf [X+r1]->f1   // I is in r1
    mulf f0,f1->f2   // A is in f0
    ldf [Y+r1]->f3
    addf f2,f3->f4
    stf f4->[Z+r1}
    addi r1,4->r1
    blti r1,4096,L1
```

- Instruction-level parallelism (ILP) - fine grained
  - Loop unrolling with static scheduling –or– dynamic scheduling
  - Wide-issue superscalar (non-)scaling limits benefits
- Thread-level parallelism (TLP) - coarse grained
  - Multicore
- Can we do some "medium grained" parallelism?

Data-Level Parallelism

- **Data-level parallelism (DLP)**
  - Single operation repeated on multiple data elements
    - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point "vector registers"
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector or Vector+Scalar
      - addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!
  - ALUs were expensive, so one operation per cycle (not parallel)

Vector ISA Extensions (SIMD)

- Extend ISA with vector storage …
  - **Vector register**: fixed-size array of FP/int elements
  - **Vector length**: For example: 4, 8, 16, 64, …
- … and example operations for vector length of 4
  - Load vector: `ldf.v [X+r1]->v1`

    `ldf [X+r1+0]->v1`$_0$
    `ldf [X+r1+1]->v1`$_1$
    `ldf [X+r1+2]->v1`$_2$
    `ldf [X+r1+3]->v1`$_3$
  - Add two vectors: `addf.vv v1,v2->v3`

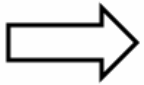
    `addf v1`$_i$`,v2`$_i$`->v3`$_i$ `(where i is 0,1,2,3)`
  - Add vector to scalar: `addf.vs v1,f2,v3`

    `addf v1`$_i$`,f2->v3`$_i$ `(where i is 0,1,2,3)`
- Today's vectors: short (128-512 bits), but fully parallel

Example Use of Vectors – 4-wide

```
ldf [X+r1]->f1
mulf f0,f1->f2
ldf [Y+r1]->f3
addf f2,f3->f4
stf f4->[Z+r1]
addi r1,4->r1
blti r1,4096,L1
```

7x1024 instructions



```
ldf.v [X+r1]->v1
mulf.vs v1,f0->v2
ldf.v [Y+r1]->v3
addf.vv v2,v3->v4
stf.v v4,[Z+r1]
addi r1,16->r1
blti r1,4096,L1
```

7x256 instructions
(4x fewer instructions)

- Operations
  - Load vector: `ldf.v [X+r1]->v1`
  - Multiply vector to scalar: `mulf.vs v1,f2->v3`
  - Add two vectors: `addf.vv v1,v2->v3`
  - Store vector: `stf.v v1->[X+r1]`
- Performance?
  - Best case: 4x speedup
  - But, vector instructions don't always have single-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

Vector Datapath & Implementation

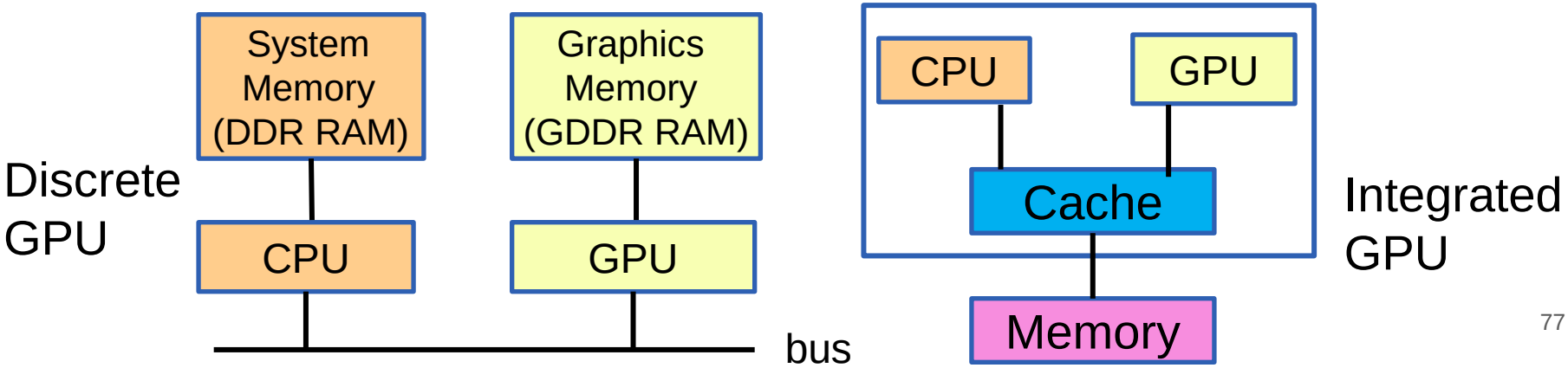
- Vector insn. are just like normal insn… only "wider"
  - Single instruction fetch (no extra N^2 checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate floating point unit (same as superscalar)
  - Wide bypass (avoid N^2 bypass problem)
  - Wide cache read & write (single cache tag check)
- Execution width (implementation) vs vector width (ISA)
  - Example: Pentium 4 and "Core 1" executes vector ops at half width
  - "Core 2" executes them at full width
- Because they are just instructions…
  - …superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle

Vector ISAs

- x86
  - Intel and AMD: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
  - currently: AVX 512 offers 512b vectors
- PowerPC
  - AltiVEC/VMX: 128b
- ARM
  - NEON: 128b
  - Scalable Vector Extensions (SVE): up to 2048b
    - implementation is narrower than this!
    - makes vector code portable
- RV
  - vector length & element size stored in registers (not opcodes)
  - allows for a high degree of flexibility

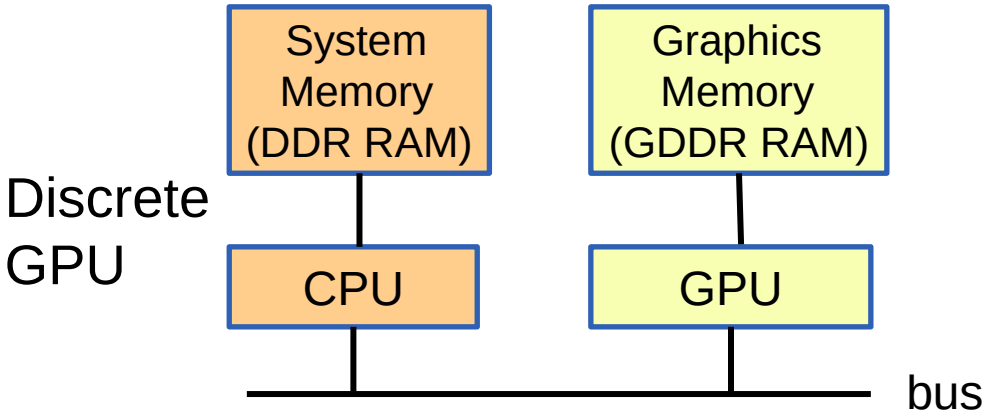
What is GPU?

- GPU = Graphics Processing Units
- Accelerate computer graphics rendering and rasterization
- Highly programmable (OpenGL, OpenCL, CUDA, HIP etc..)
- Why does GPU use GDDR memory?
  - DDR RAM -> low latency access, GDDR RAM -> high bandwidth



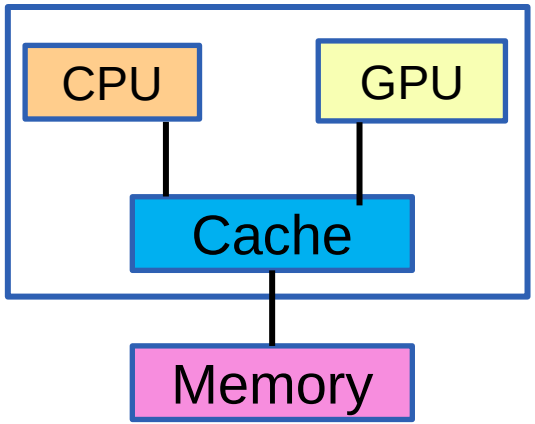
Discrete GPU

- A (PCIe) bus connecting the CPU and GPU
- Separate DRAM memory spaces
  - CPU (system memory) and the GPU (device memory)
- DDR for CPU vs. GDDR for GPU
  - CPU DRAM optimizes for low latency access
  - GPU DRAM is optimized for high throughput



Integrated GPU

- Have a single DRAM memory space
- Often found on low-power mobile devices
  - Ex. AMD APU
  - Private cache -> cache coherence



Integrated GPU

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4	4.2 GHz	DDR4 RAM	$385	~540 GFLOPs F32
GPU (Nvidia RTX 3090 Ti)	10496	1.7 GHz	DDR6 24 GB	$1499	36 TFLOPs F32

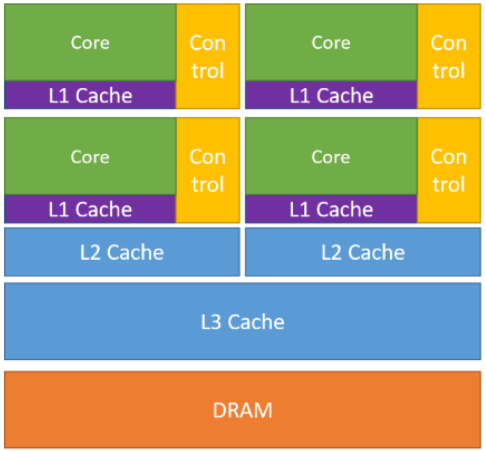
CPU: A **small** number of **complex** cores, the clock speed of each core is high, great for sequential tasks
GPU: A **large** number of **simple** cores, the clock speed of each core is low, great for parallel tasks

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htm

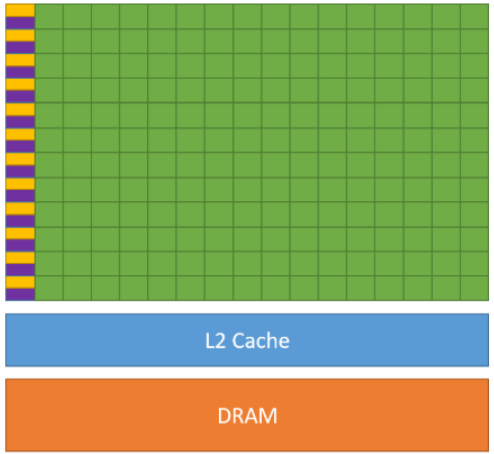
Why do we use GPU for computing ?

- What is difference between CPU and GPU?
  - GPU uses a large portion of silicon on the computation against CPU
  - GPU (2nJ/op) is more energy-efficient than CPU (200 pJ/op) at peak performance
  - Need to map applications on the GPU carefully (Programmers' duties)

CPU

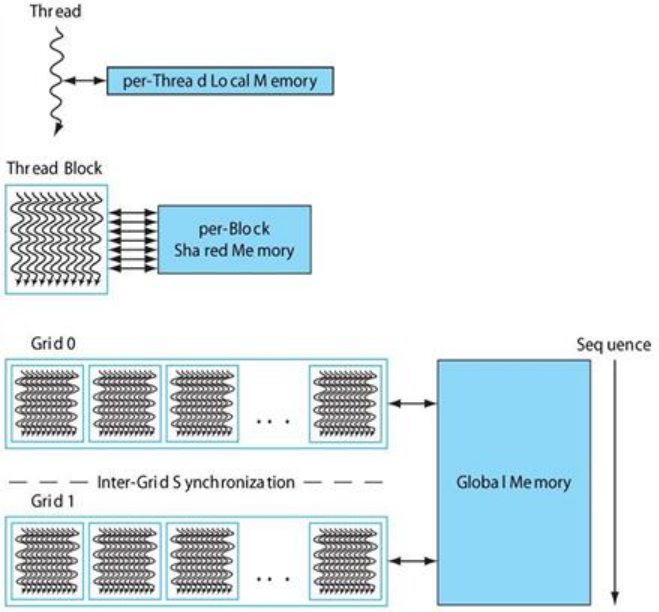


GPU



GPU Programming

- Software GPU Thread Model (CUDA)
  - Single-program multiple data (SPMD)
  - Each thread has local memory
  - Parallel threads packed in blocks
    - Access to per-block shared memory
    - Synchronize with barrier
  - Grids include independent groups



GPU Programming

- In SAXPY example
  - CUDA code launches 256 threads per block
    - Thread = 1 iteration of scalar loop (1 element in vector loop)
    - Block = body of vectorized loop (with VL = 256 in this ex.)
    - Grid = vectorizable loop

C Code

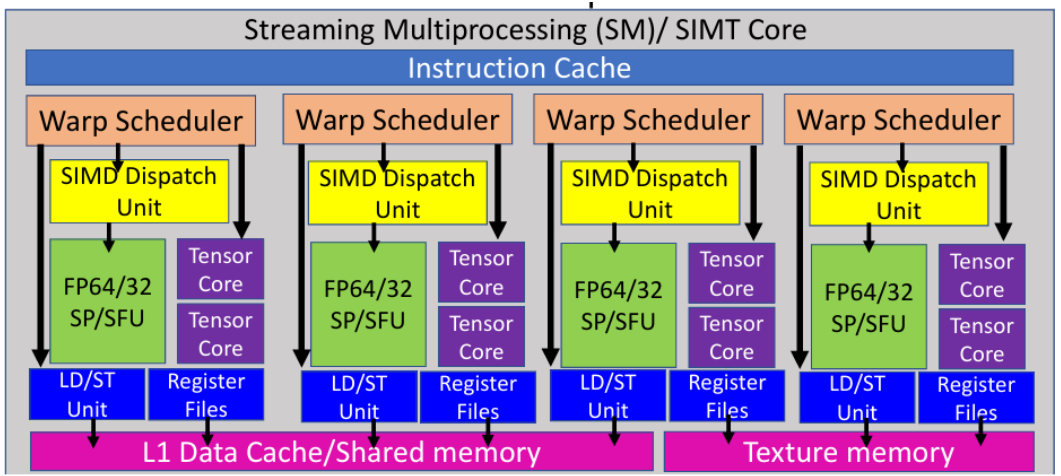
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
        for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

CUDA Code

```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

GPU Architecture

- 15 SMX processors, shared L2, 6 memory controllers

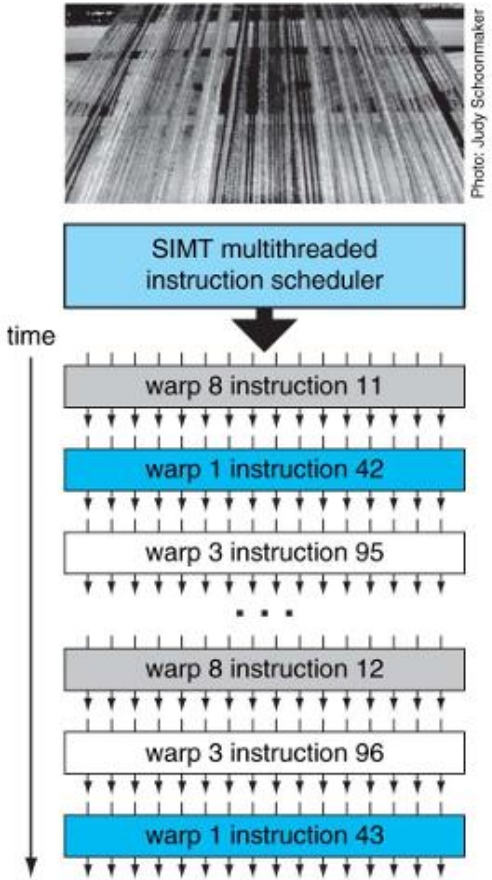


GPU Architecture

- Cores are
  - Multithreaded
  - Data parallel
- Capabilities
  - 64K registers
  - 192 simple cores
    - Integer and SP FPU
  - 64 DP FPUs
- Scheduling
  - 4 warp schedulers, 2 instruction dispatch per warp

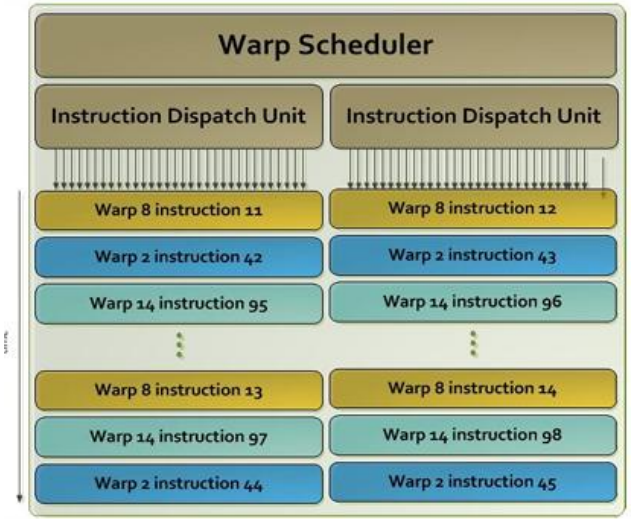
GPU Architecture

- All threads can be independent
  - HW implements zero-overhead switching
- 32 threads are packed in warps
  - **Warp**: set of parallel threads that execute the same instruction-> data parallelism
  - 1 warp instruction keeps cores busy for multiple cycles
- SW thread blocks mapped to warps
  - When HW resources are available



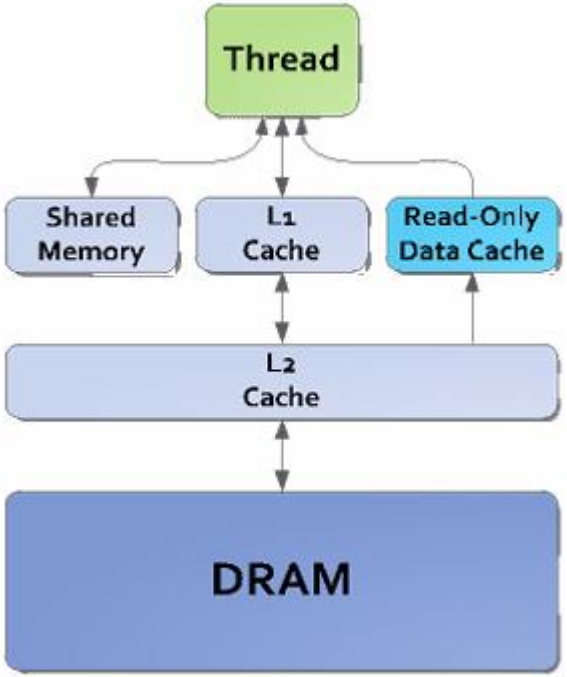
GPU Architecture

- 64 warps per SMX
- 32 threads per warp
  - 64K registers/SMX
  - Up to 255 registers per threads (8 warps)
- Scheduling
  - 4 schedulers select 1 warp per cycle
  - 2 independent instructions issued per warp (double-pumped FUs)
  - Total bandwidth = 4 x 2 x 32 = 256 ops per cycle



GPU Architecture

- Each SMX has 64KB of memory
  - Split between shared mem and L1 cache
  - 256 Bytes per access
  - 48KB read-only data cache
  - 1.5MB shared L2
    - Supports synchronization operations (atomicCAS, atomicADD …)
  - Throughput-oriented main memory
    - GDDRx standards



Conclusion

- Instruction-Level Parallelism (ILP)
  - Pipelining, super-scalar processor
- Thread-Level Parallelism (TLP)
  - Hardware multi-threading
- Data-Level Parallelism (DLP)
  - SIMD, Vector processor, GPU