

# CPEN 411: Computer Architecture

## Slide Set #13: Multicore Concepts

Instructor: Mieszko Lis

Original Slides: Tor Aamodt

# Introduction to Slide Set 13

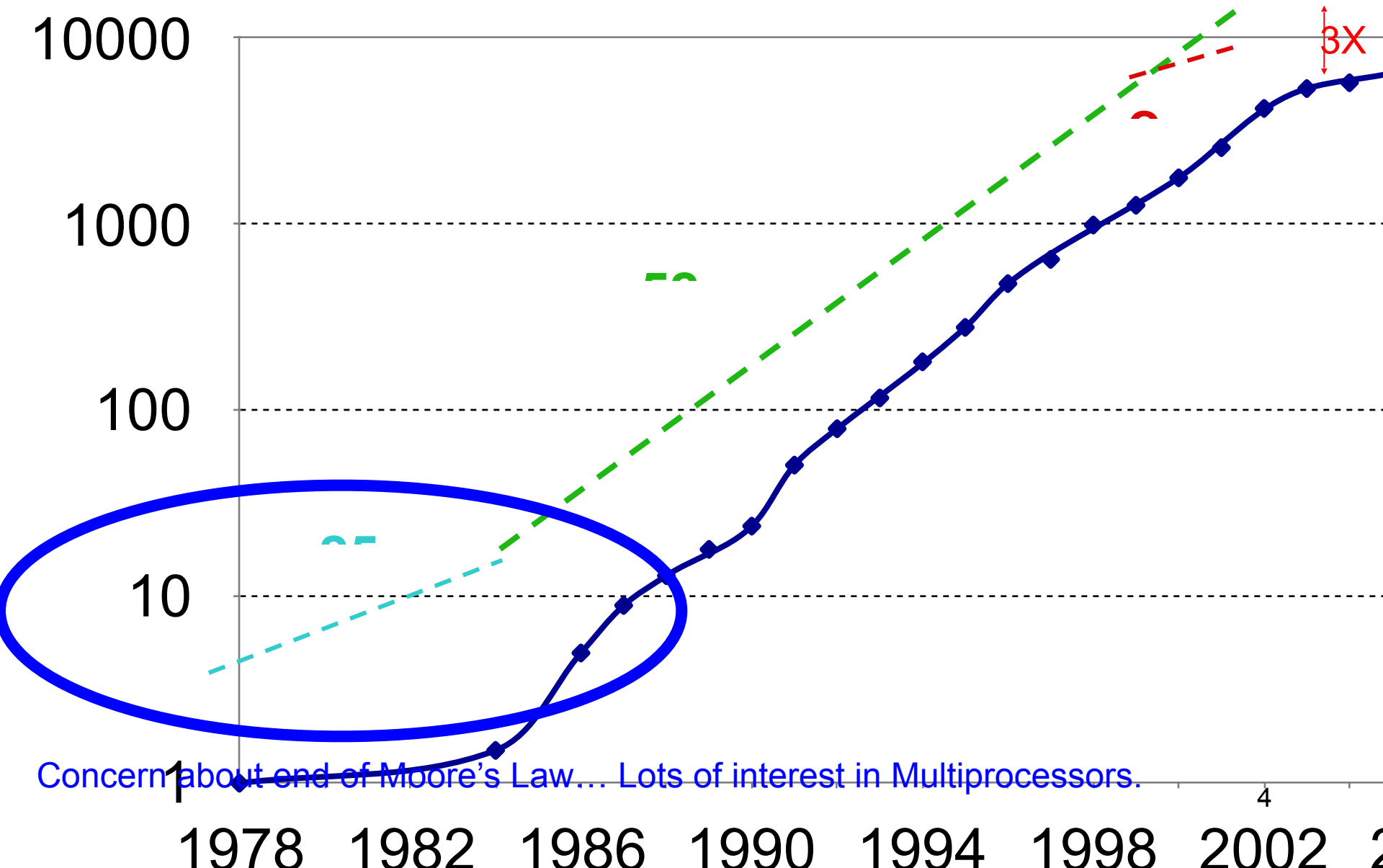
- So far in the course we have learned how an individual processor core works. In the next few slide sets we focus on how to connect cores together.
- This slide set introduces the benefits and challenges of having multiple cores.

# Learning Objectives

By the time we finish talking about this slide set in lectures you should be able to:

- explain the challenges of obtaining performance using parallel processing
- contrast multithreading, fine-grained multithreading, and simultaneous multithreading (“hyperthreading”)
- discuss memory consistency

# Why (Explicitly) Parallel Computing?



# Challenges of Parallel Processing

- First challenge is % of program inherently sequential
- Suppose we want to obtain an 80X speedup from 100 processors (versus 1 processor). What fraction of original program can be sequential?
  - a. 10%
  - b. 5%
  - c. 1%
  - d.  $<1\%$



# Amdahl's Law Answers

# Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$



# Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

# Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times \left( \left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100} \right) = 1$$

# Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times \left( \left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100} \right) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

# Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times \left( (1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100} \right) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

# Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory, all local accesses hit memory hierarchy and base CPI is 0.5. (Remote access =  $200/0.5 = 400$  clock cycles.)
- What is performance impact if 0.2% instructions involve remote access?
  - a. 1.5X
  - b. 2.0X
  - c. 2.5X

# Solution

# Solution

- $CPI = \text{Base CPI} +$   
     $(\text{Remote request rate}$   
     $\times \text{Remote request cost})$
- $CPI = 0.5 + 0.2\% \times 400 = 0.5 + 0.8 = 1.3$

# Solution

- $CPI = \text{Base CPI} +$   
     $(\text{Remote request rate}$   
     $\times \text{Remote request cost})$
- $CPI = 0.5 + 0.2\% \times 400 = 0.5 + 0.8 = 1.3$

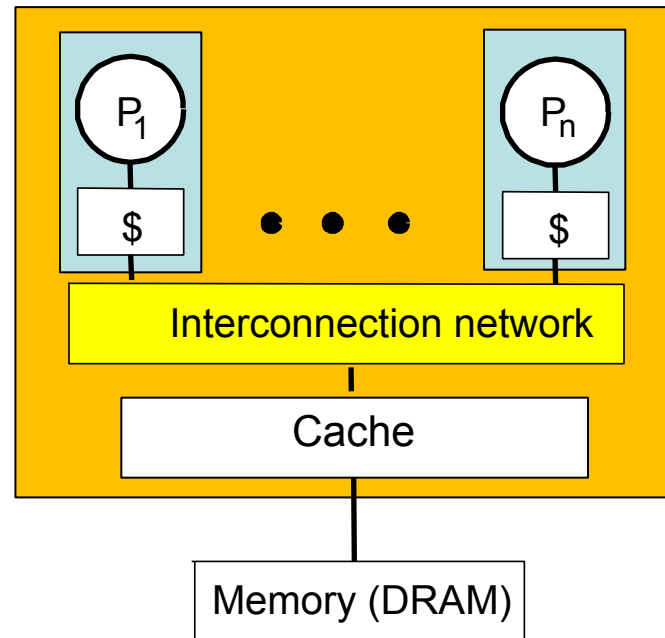


# Multicore Processors

Each  $P_i$  is a processor like those studied earlier in the course. The boxes labeled “\$” are caches.

Multiple processor cores on a single chip.

Processors are connected via an interconnection network (bus, crossbar, ring, mesh).



# How to program?

- Easiest: Multiprogramming
  - Separate single thread applications
  - Used today for “Cloud Computing”
- For some programs: Parallelizing Compiler
  - Input C/C++/Fortran; Output parallel program
- General: Parallel APIs / Languages
  - APIs: Pthreads, OpenMP, MPI
  - Languages: CUDA, OpenCL, High Performance Fortran, Erlang, Sisal

# POSIX threads (pthreads)

```
#define N (1024*1024)
double data[N], sum[2];

void *foo() { int i; for(i=0;i<N/2;i++) sum[0] += data[i]; }
void *bar() { int i; for(i=0;i<N/2;i++) sum[1] += data[i+N/2]; }

int main(int argc, char *argv[])
{
    pthread_t f, b;

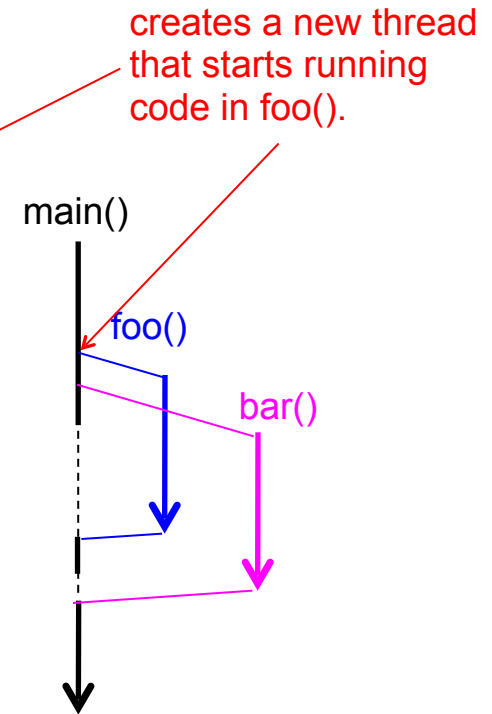
    // load data with values here...

    pthread_create(&f, NULL, foo, NULL);
    pthread_create(&b, NULL, bar, NULL);

    pthread_join(f, NULL);
    pthread_join(b, NULL);

    double total = sum[0]+sum[1];

    // do something with total here
    return 0;
}
```

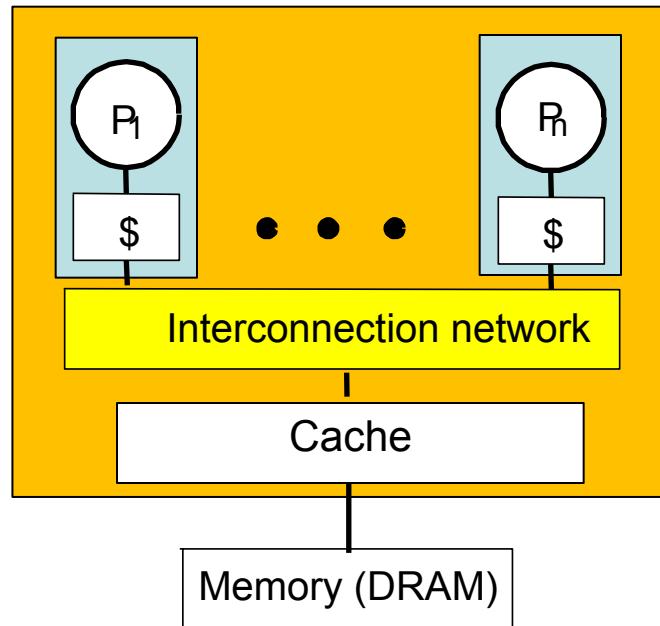


Create two new threads. Operating system can schedule them on different cores so they run at same time (concurrently).

**thread 0: main()**

**thread 1: foo()**

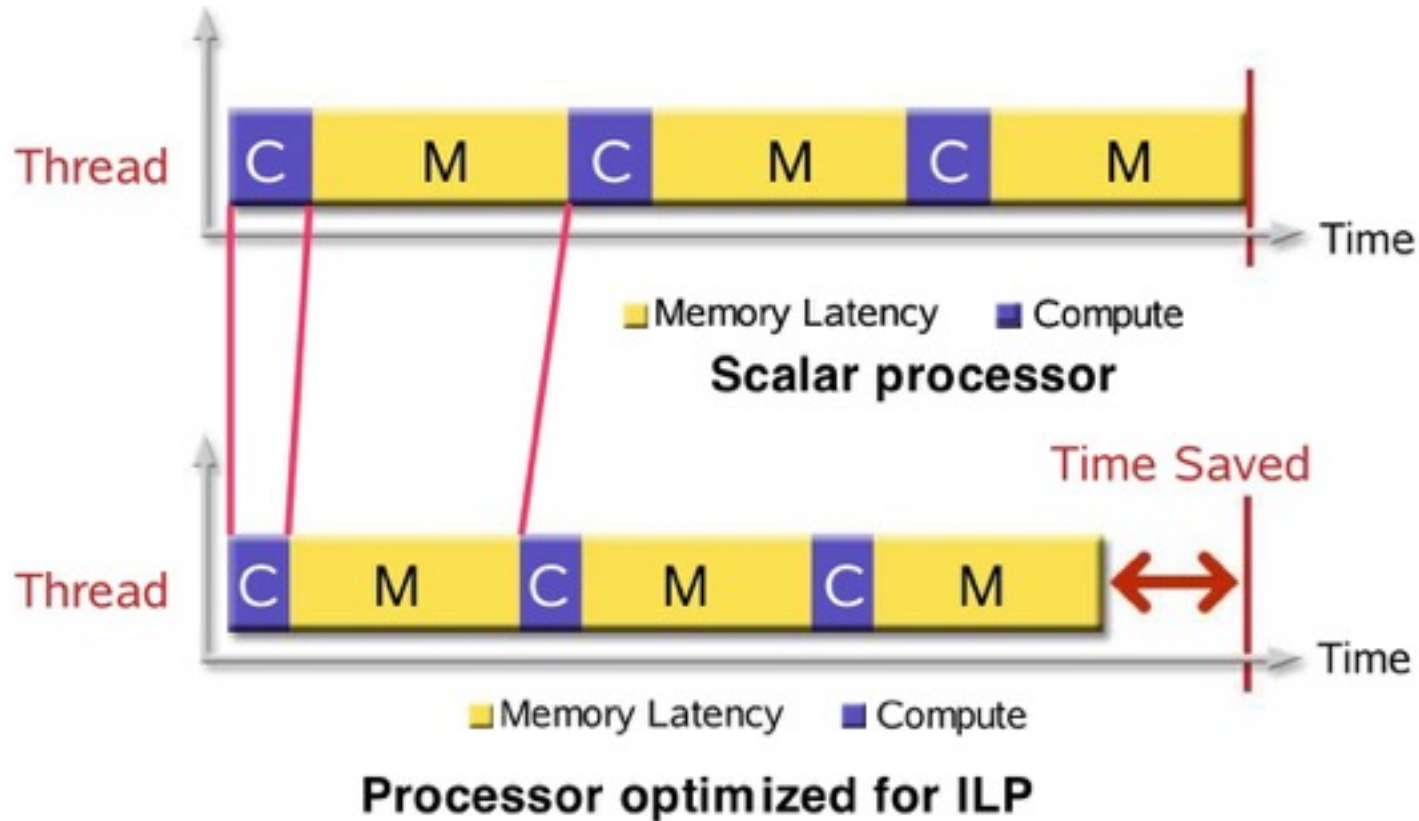
**thread 1: bar()**



# One thread per core?

- Multiple cores enable multiple threads to run at the same time if we have one thread on each core.
- However, with one thread per core, cache misses can cause each core to be idle most of the time.
- Would like to “multiplex” more than one thread per core, but an operating system context switch takes far longer than the latency of a cache miss.

# Hardware Idle Due to Memory Latency

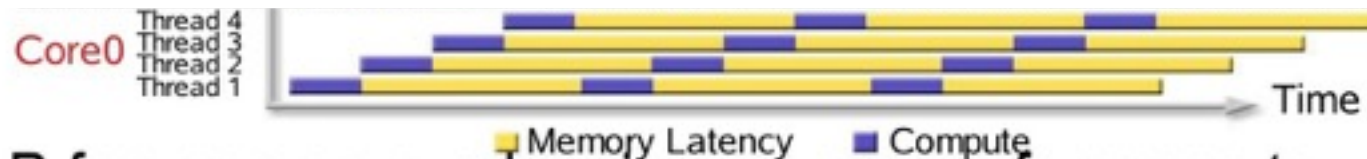


ILP reduces the compute time and overlaps computation with L2 cache hits, but memory stall time dominates overall performance

# Hardware Multithreading?

- If operating system is too slow to context switch, can we have hardware do this for us?
- Answer: Yes, but there is some area and complexity overhead.

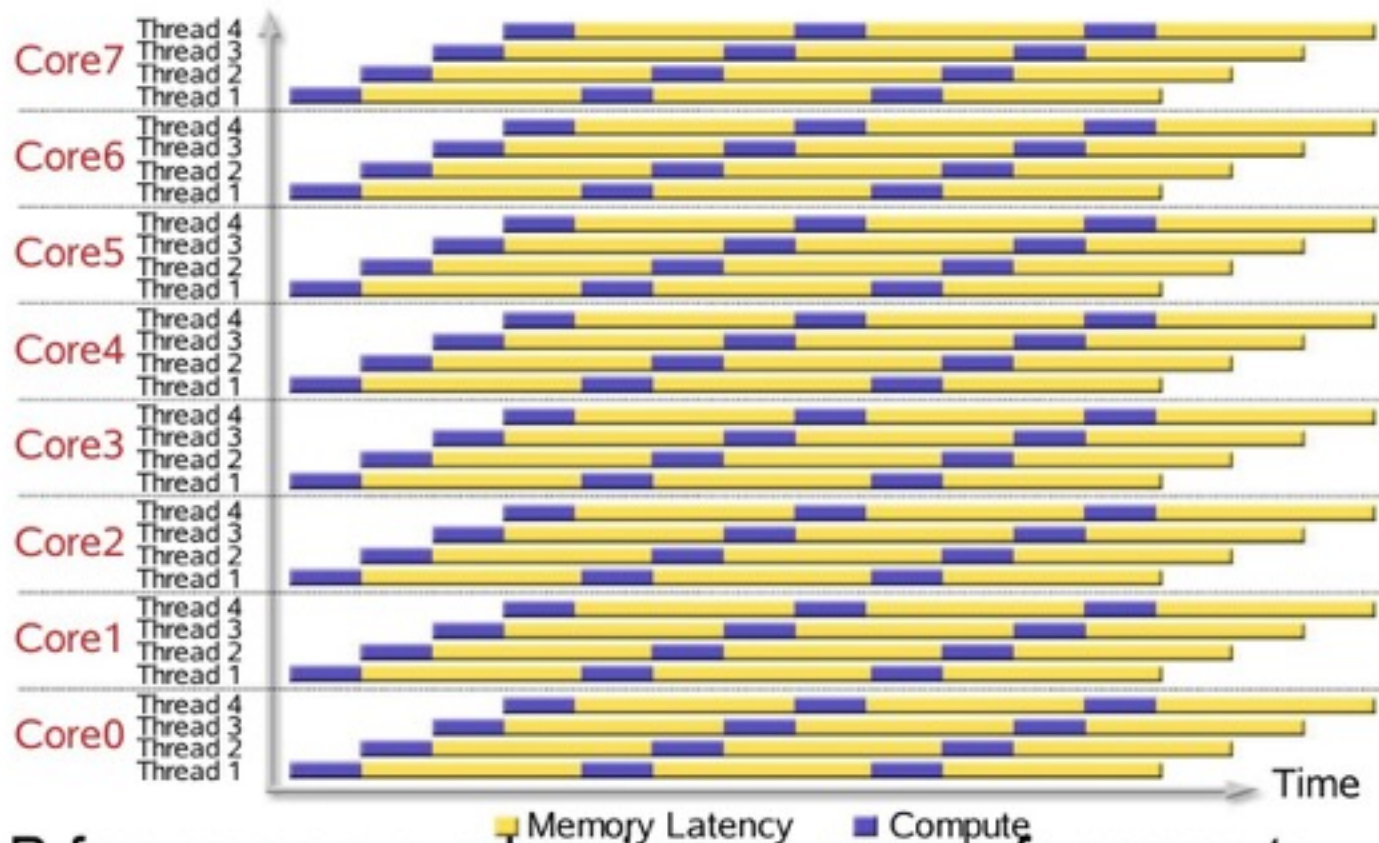
# Fine-Grained Multithreading



TLP focuses on overlapping memory references to improve throughput; needs sufficient memory bandwidth



# Fine-Grained Multithreading



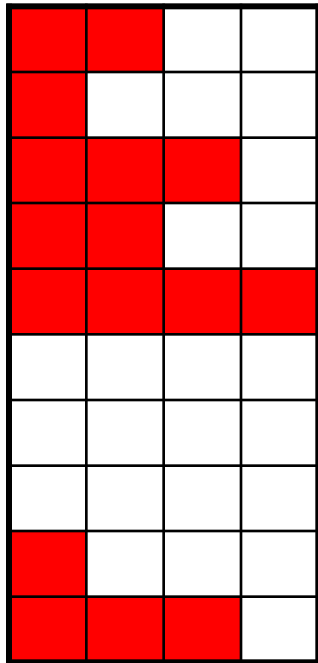
TLP focuses on overlapping memory references to improve throughput; needs sufficient memory bandwidth

Observation: Superscalar processors that support multiple issue seldom sustain peak instructions per cycle

# Simultaneous Multithreading

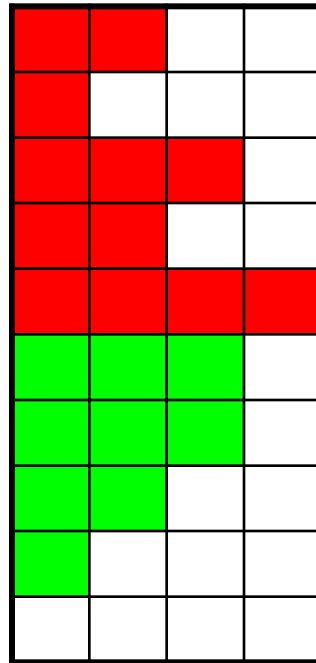
Superscalar  
=pipelining+  
multiple issue+  
Tomasulo+ROB+  
BTB+BP

Time

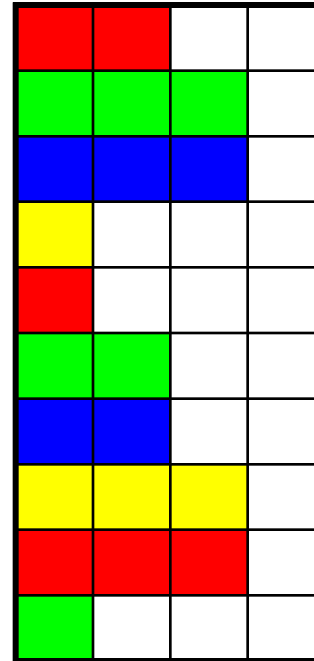


Execute Slots

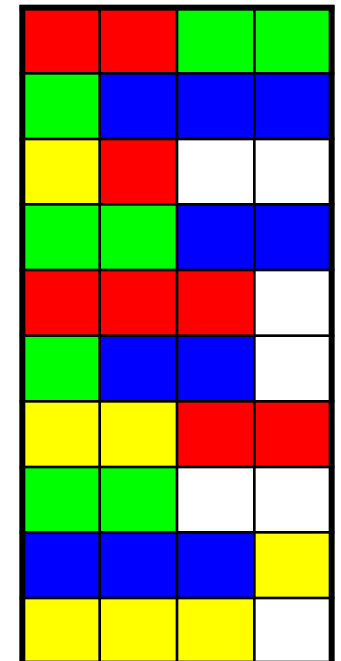
Coarse Grained  
Multithreading  
(CGMT)  
Uniprocessor w/  
Virtual Memory



Fine Grained  
Multithreading  
(FGMT)  
Ex. GPUs,  
Sun Niagara



Simultaneous  
Multithreading  
(SMT)  
Intel  
“Hyperthreading”



# Parallel Communication Models

1. Communication occurs by explicitly passing messages among the processors: **message-passing**
  - Benefit: Scalable -- systems with 100K processors use this
  - Drawback: Programmer effort to divide up data among processors and explicitly communicate changes to data.
  
2. Communication occurs through a shared address space via load and store instructions: **shared memory**
  - Benefit: Less initial work for programmer
  - Drawback: Not as scalable. Effort to obtain high performance

# Shared Memory

- We focus on shared memory as this is what you see in today's microprocessors (e.g., Intel Core i5, i7; and mobile system-on-chips such as Snapdragon).
- Communication occurs via memory—from earlier example:
  - foo() writes sum[0]
  - bar() writes sum[1]
  - main() reads sum[0] and sum[1] and adds them.

# Two Shared Memory Concepts

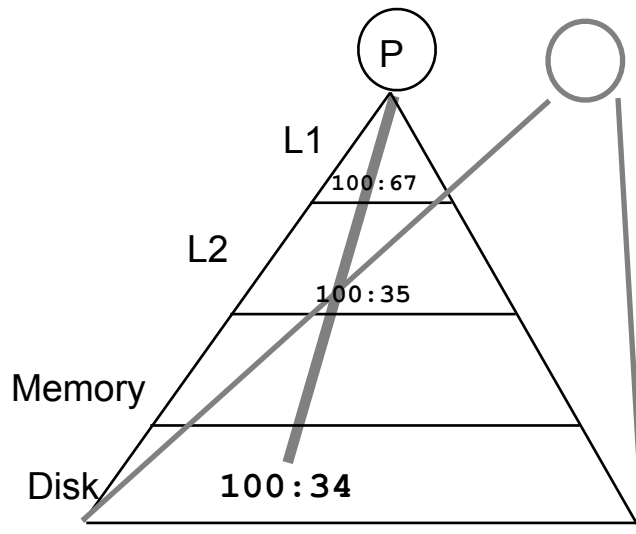
## 1. Memory Consistency Model

- Specifies ordering of updates to different locations that are allowed by hardware

## 2. Cache Coherence Protocol

- Ensures updates to any single location by one thread become visible to threads on other cores

# Intuitive “Memory Model”



Reading an address should return the “last value written to that address”

# Another Silly Analogy

- Instructor tells ECE office to change lecture location on SSC (e.g., from WESB to MCLD), then sends text message to students telling them to check new location on SSC.
- Diligent student gets text, immediately checks SSC and sees room is WESB since ECE office busy and has not yet updated SSC.
- Diligent student goes to wrong room!
- Instructor sent messages in correct order!
- Solutions?



# Memory Consistency Problem

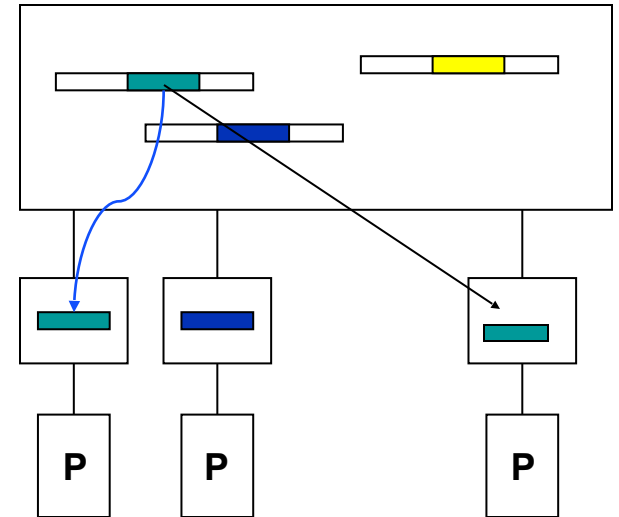
Example:

P1:	A = 0;	P2:	B = 0;
	...		...
	A = 1;		B = 1;
L1:	if (B == 0) ...	L2:	if (A == 0) ...

- Possible for both “if” statements L1 & L2 to be true?
- What if load is executed before store due to out of order execution?
- Memory consistency model specifies rules for ordering of loads and stores accessing different locations and executed in different threads

# Caches Critical for Performance

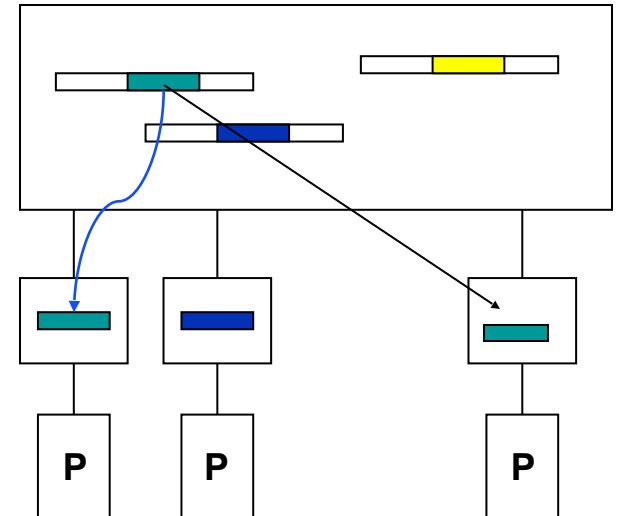
- Reduce average latency
- Reduce average bandwidth



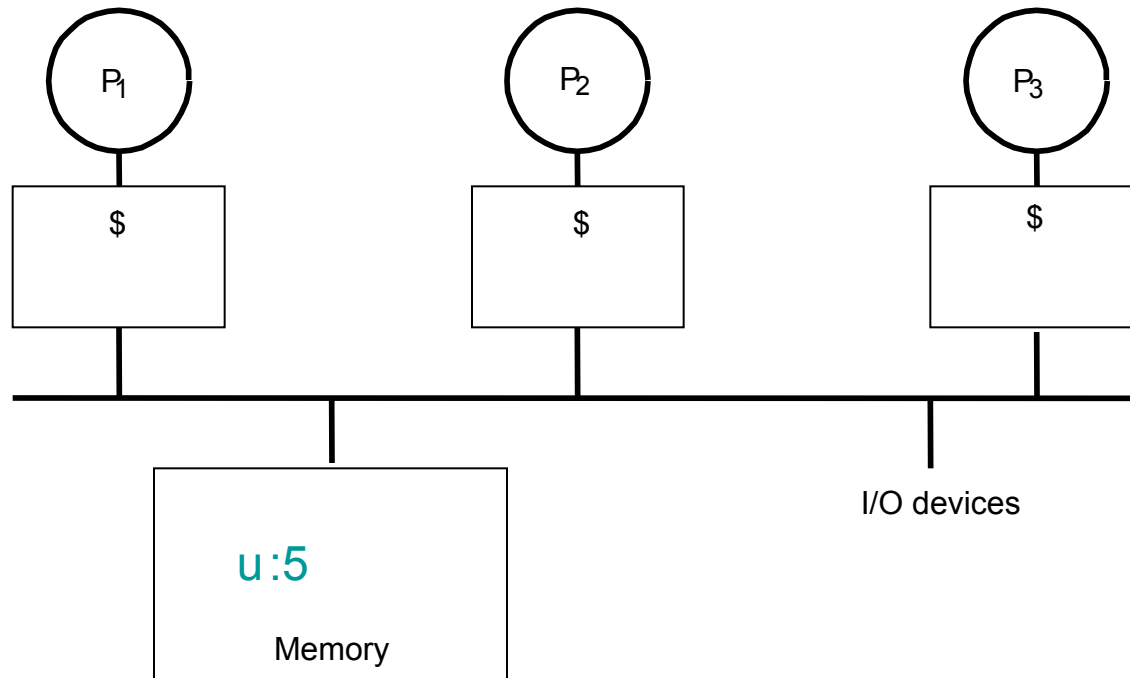
- Many processors can share data efficiently
- Problem: What happens when store & load are executed on different processors?

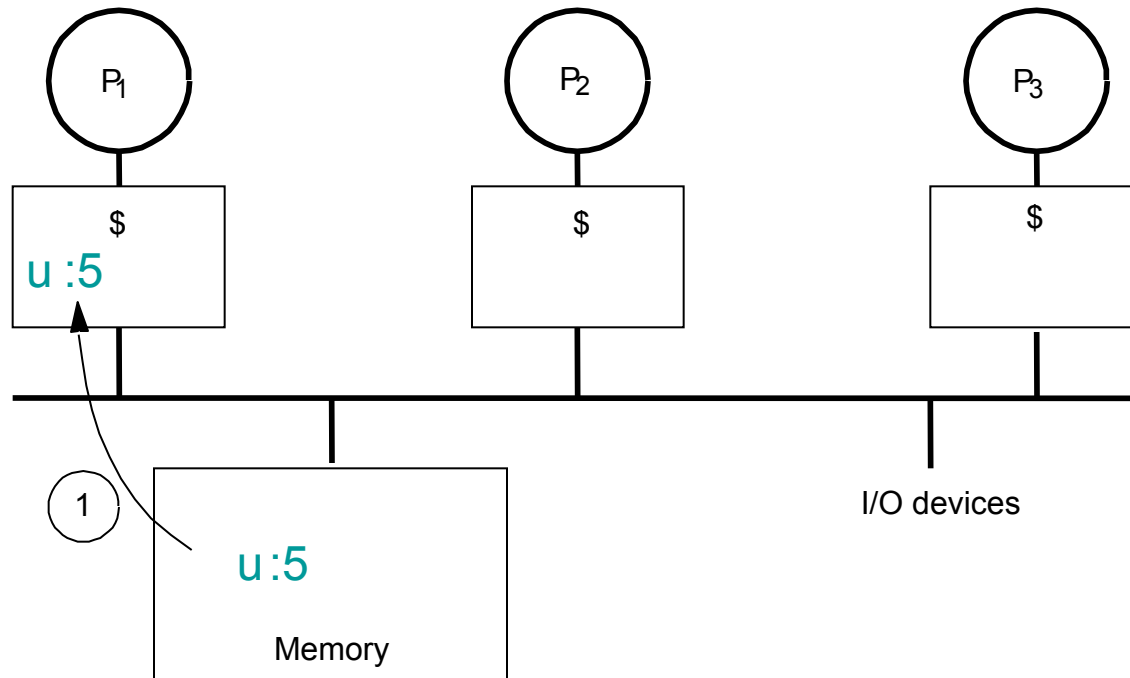
# Caches Critical for Performance

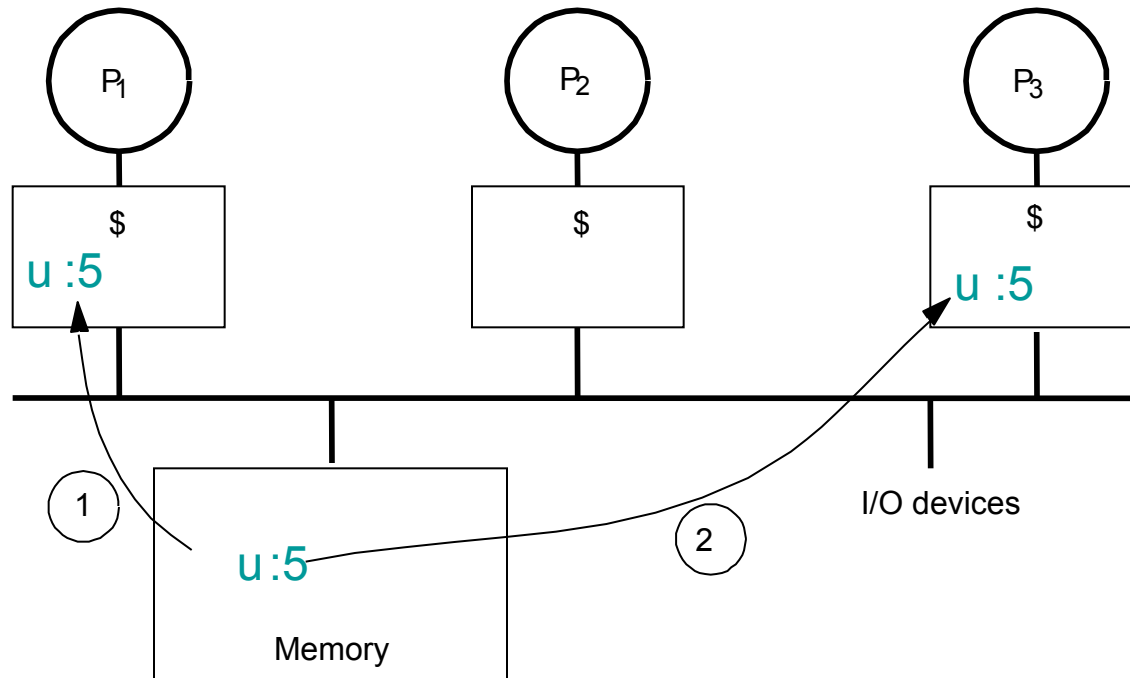
- Reduce average latency
- Reduce average bandwidth
- Data is logically transferred from producer to consumer via memory
  - store reg  $\rightarrow$  mem
  - load reg  $\leftarrow$  mem

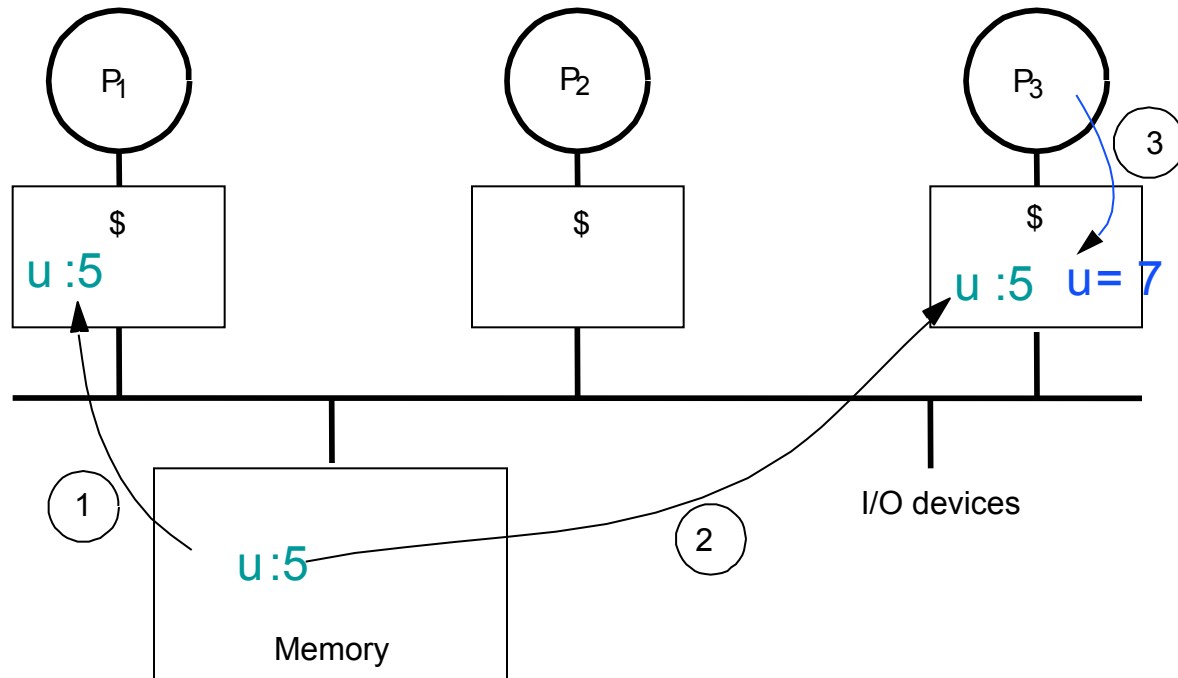


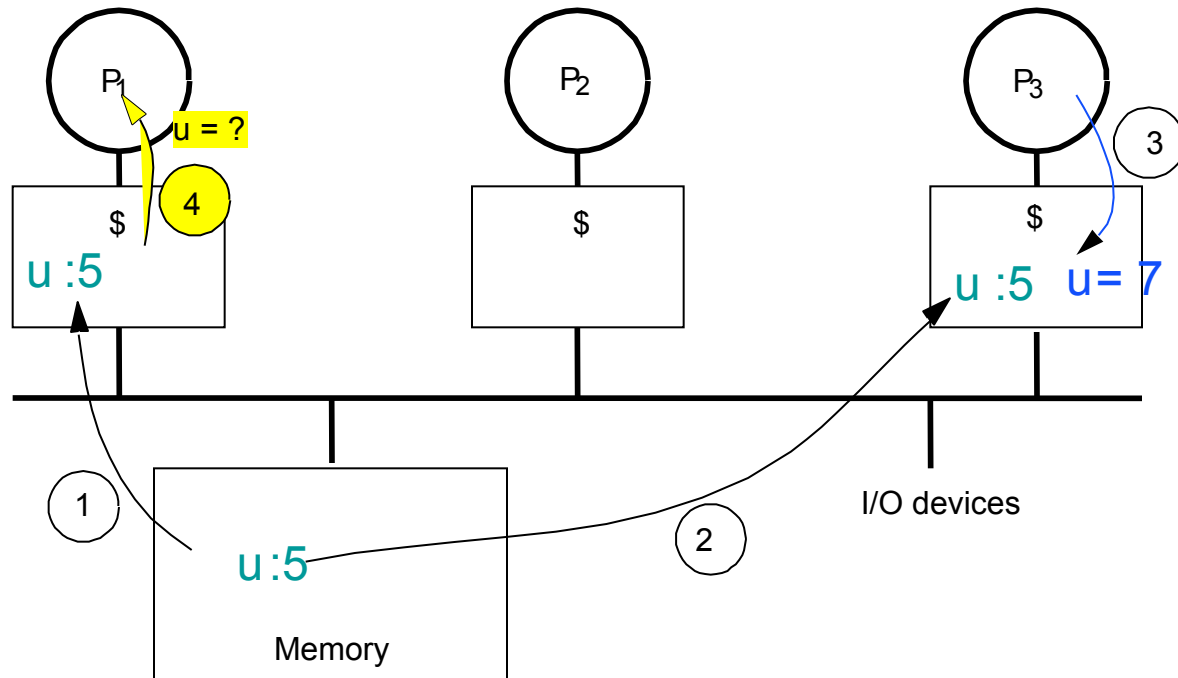
- **Many processors can share data efficiently**
- **Problem: What happens when store & load are executed on different processors?**



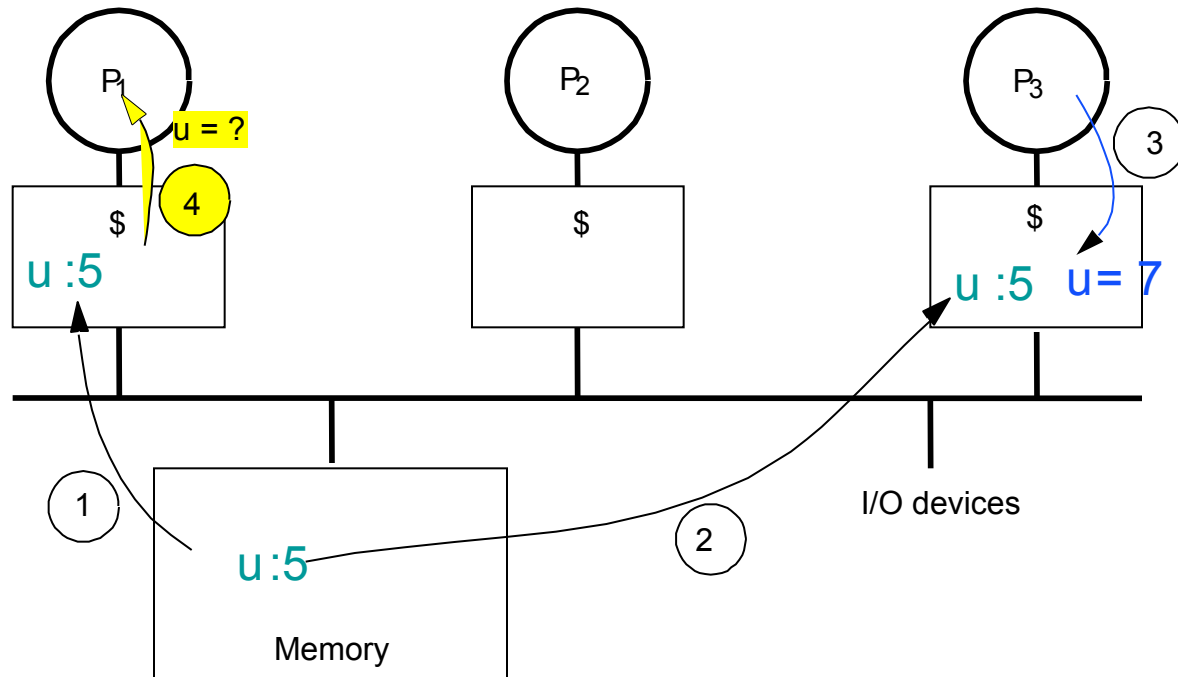










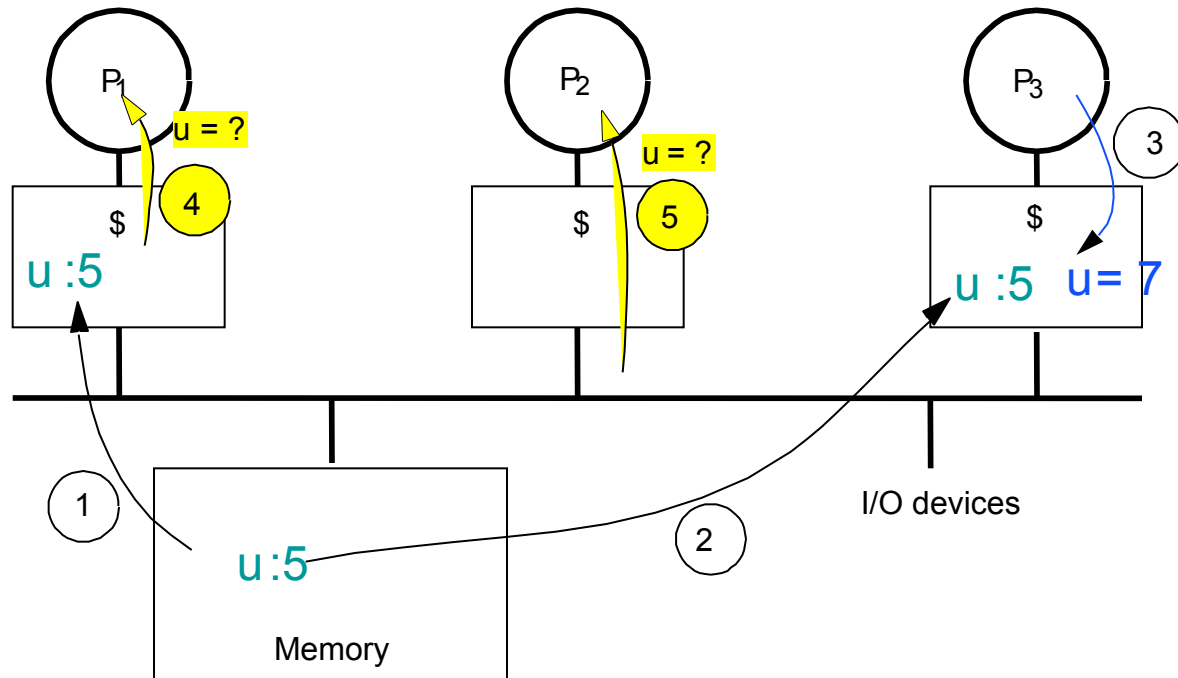


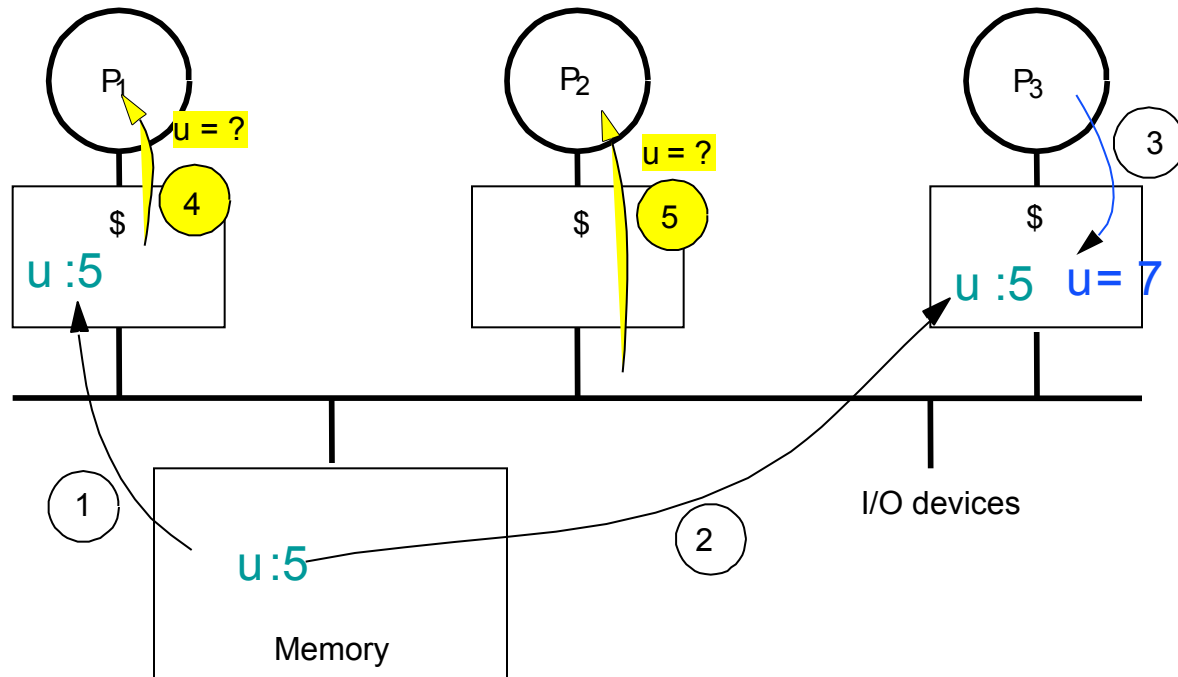
Which value of  $u$  should  $P_1$  read?

A:  $u=5$

B:  $u=7$

C: Not sure



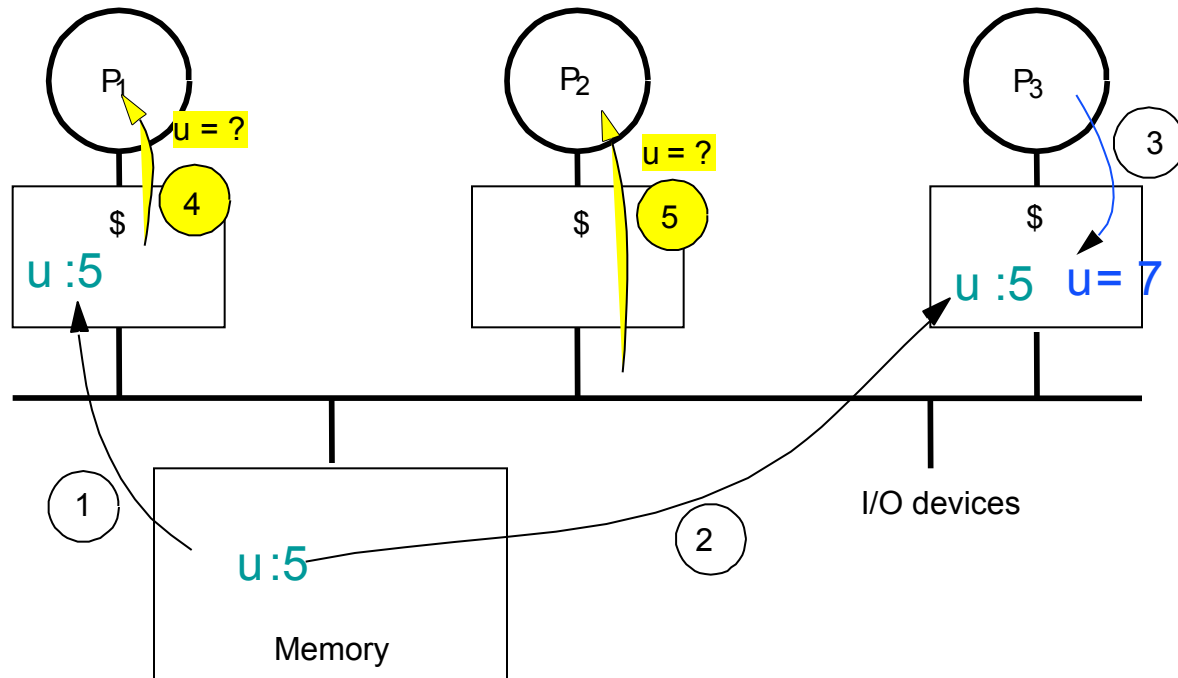


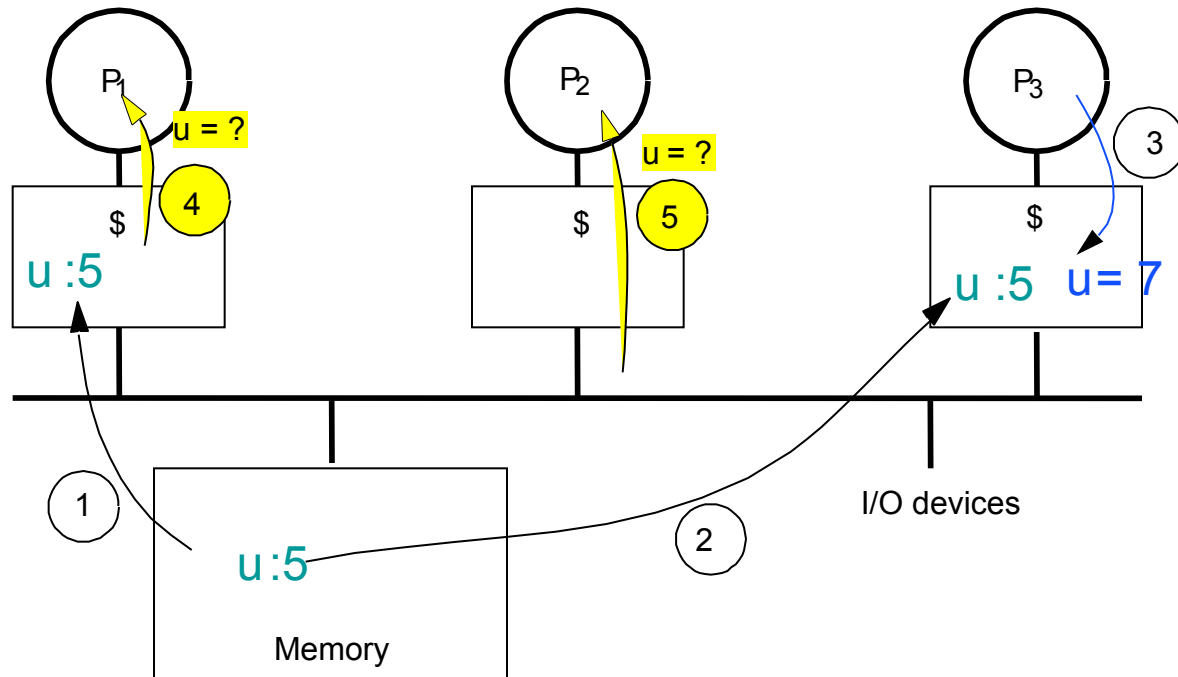
Which value of  $u$  should  $P_2$  read?

A:  $u=5$

B:  $u=7$

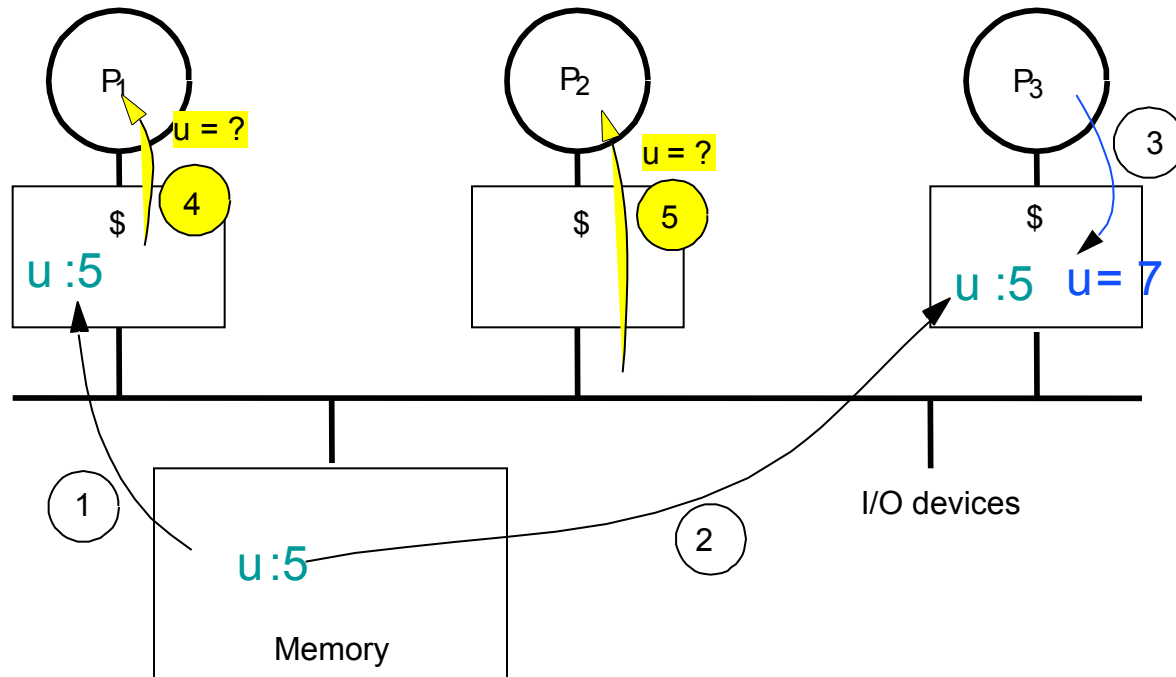
C: Not sure





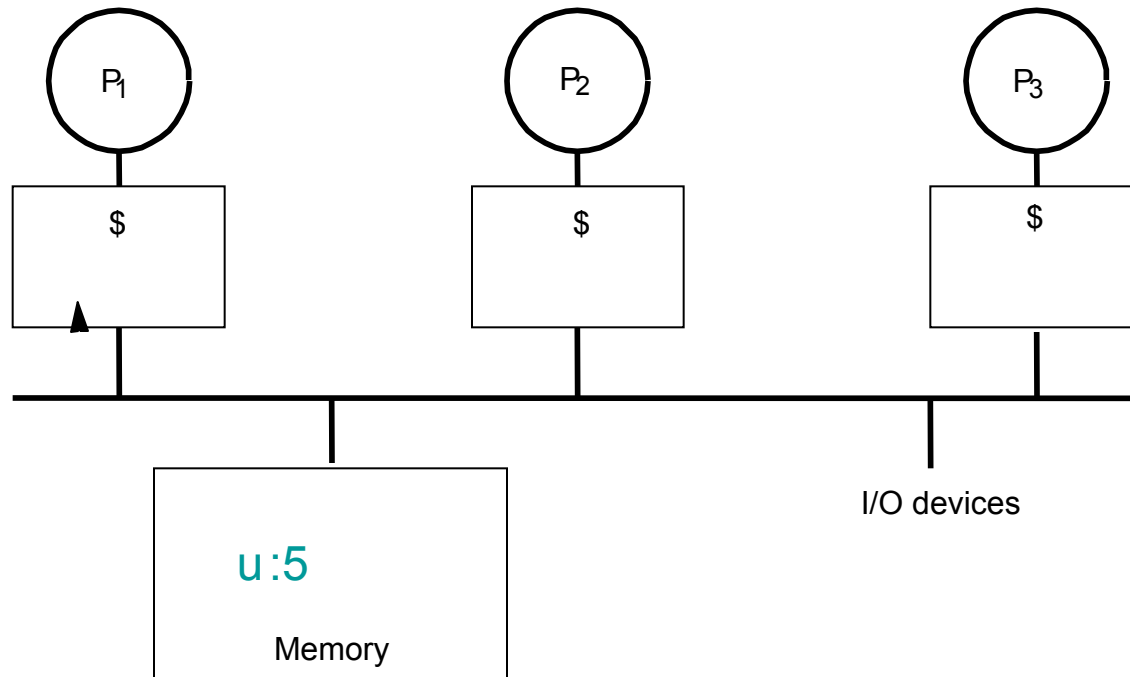
- Processors see different values for  $u$  after event 3
- With write back caches, value written back to memory depends on order of which cache writes back value first
- Unacceptable situation for programmers

# Cache Coherence Problem

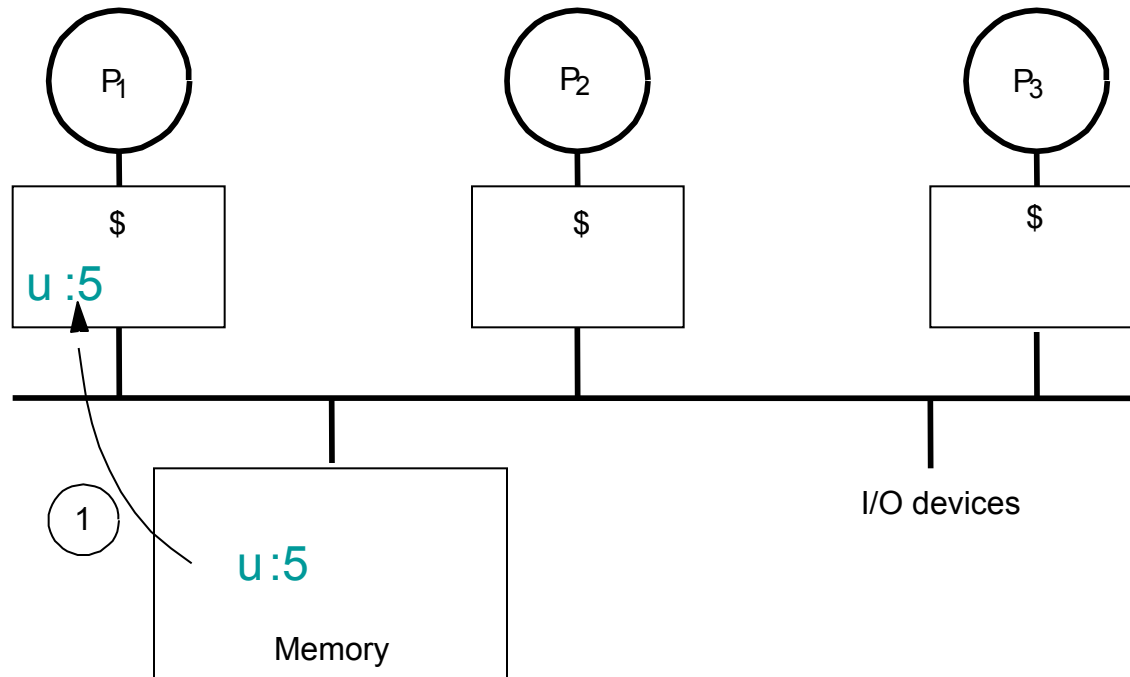


- Processors see different values for  $u$  after event 3
- With write back caches, value written back to memory depends on order of which cache writes back value first
- Unacceptable situation for programmers

# Cache Coherence Example

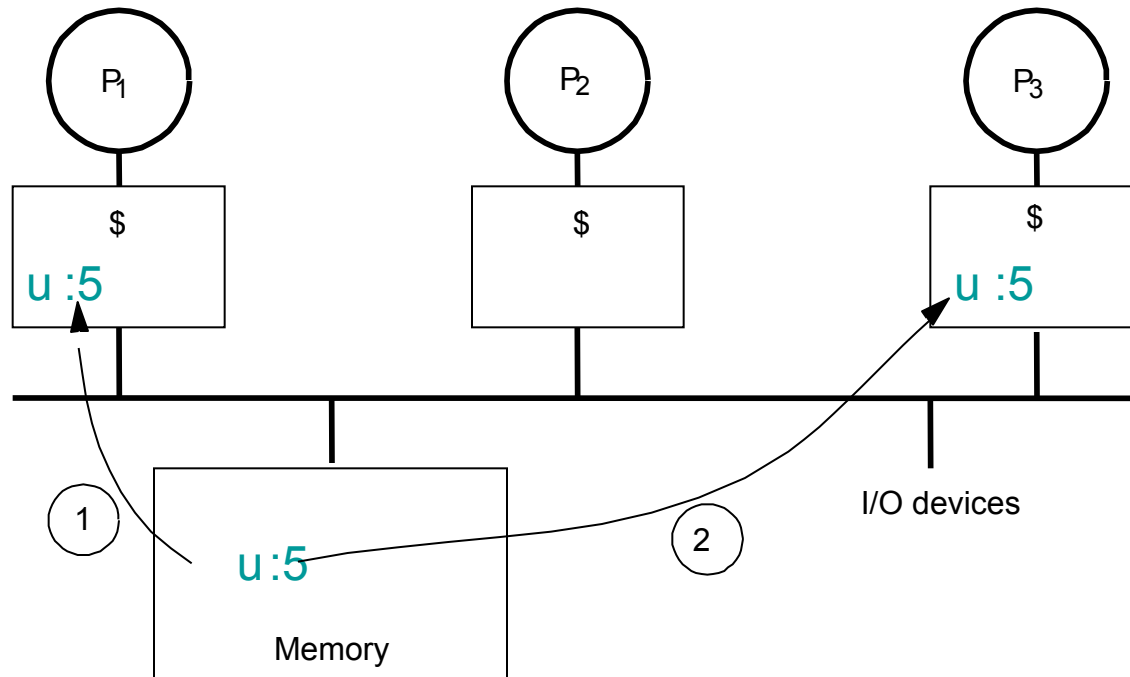


# Cache Coherence Example

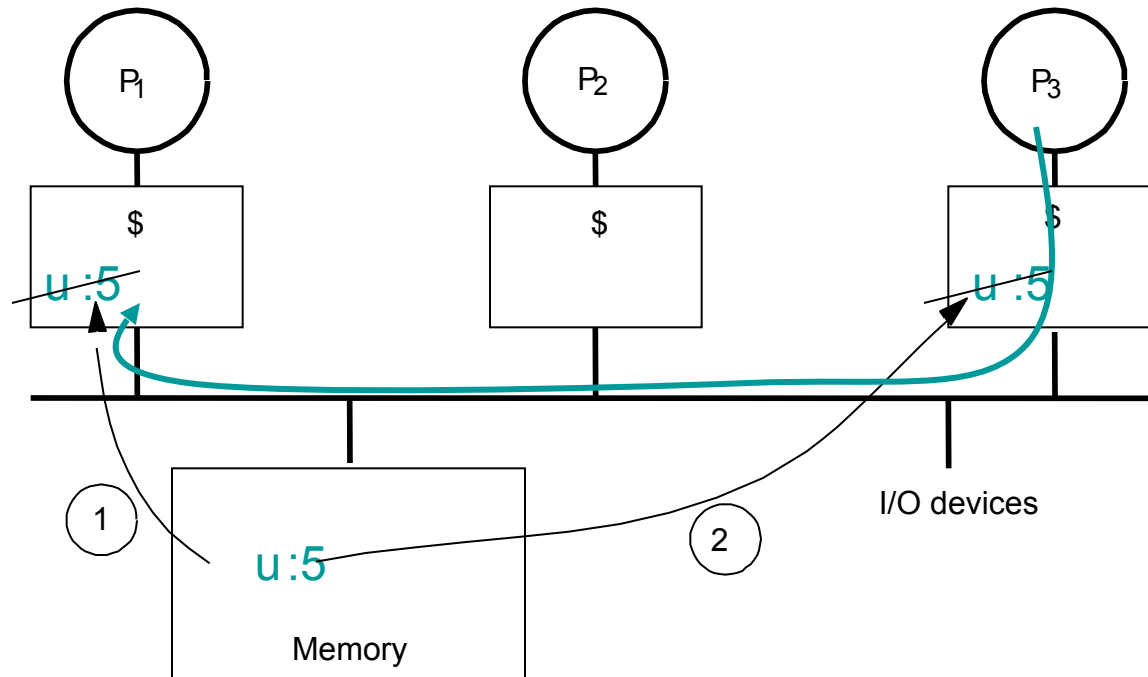




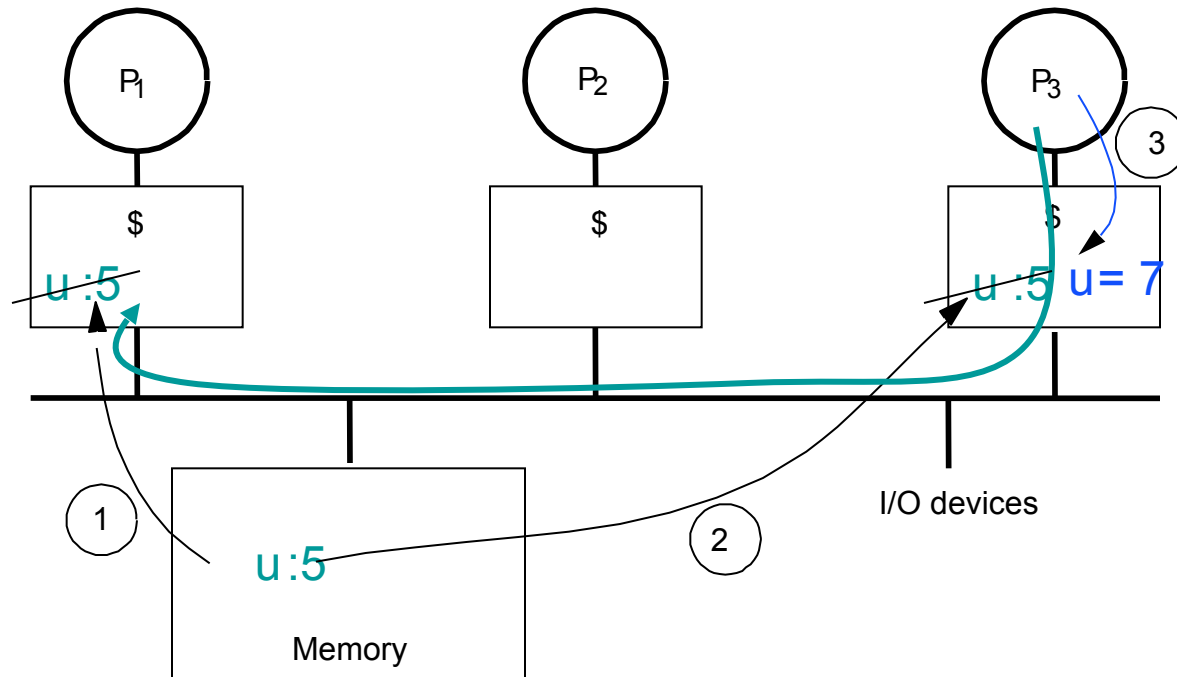
# Cache Coherence Example



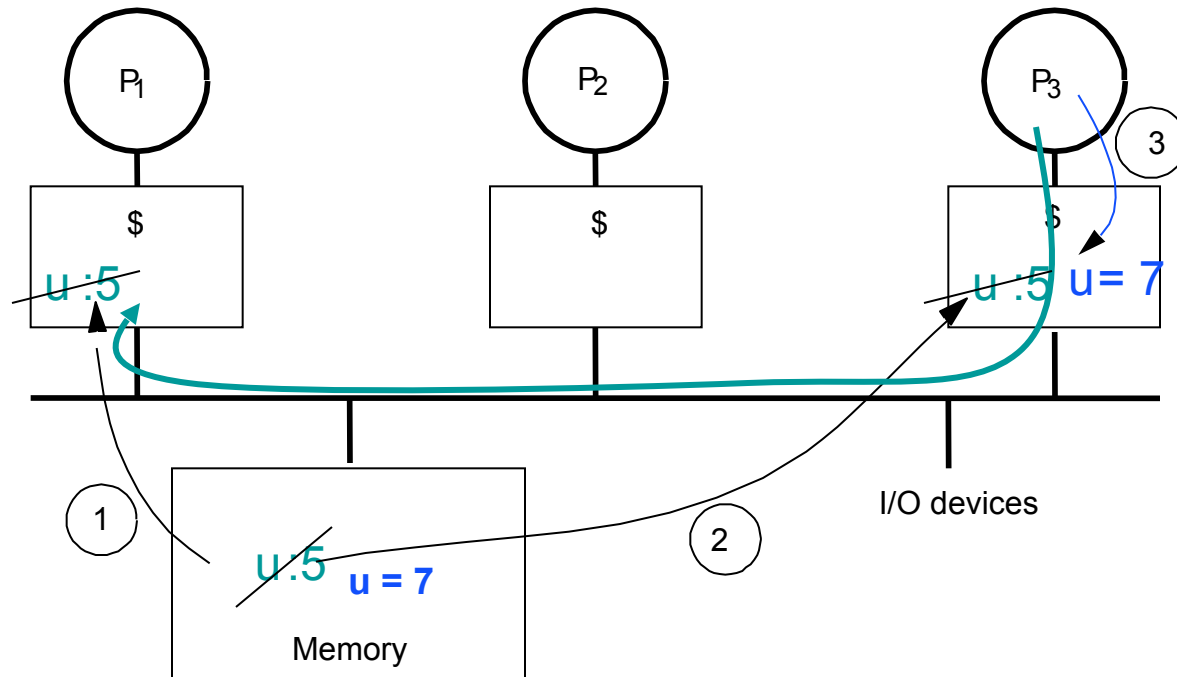
# Cache Coherence Example



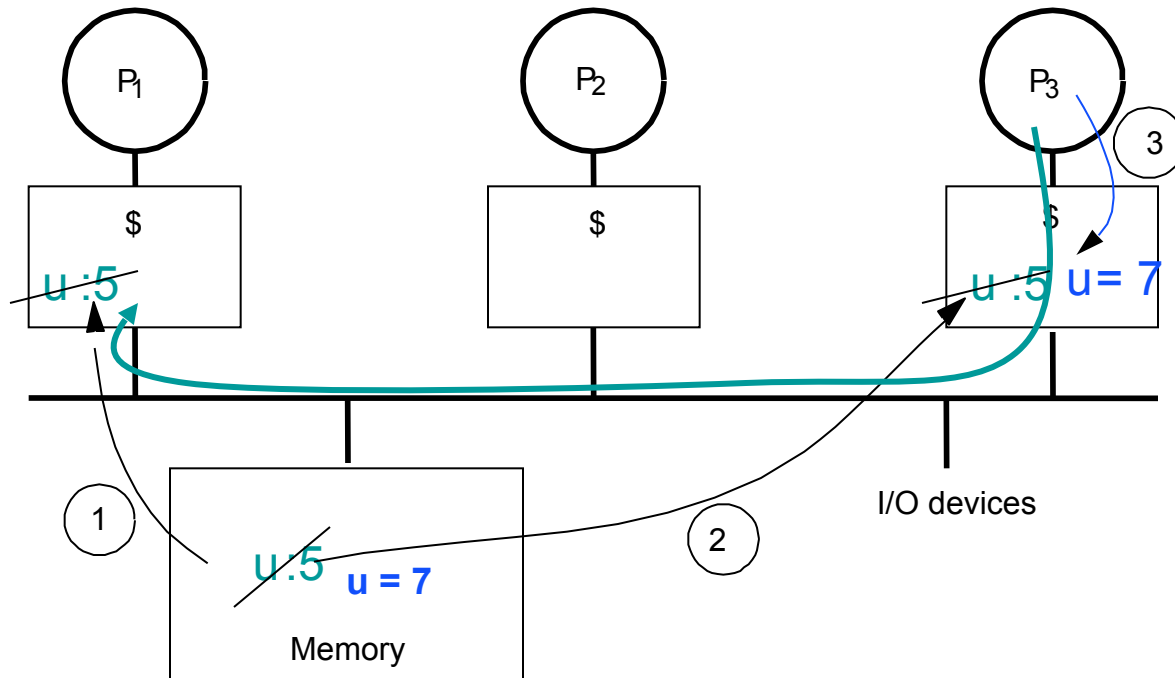
# Cache Coherence Example



# Cache Coherence Example

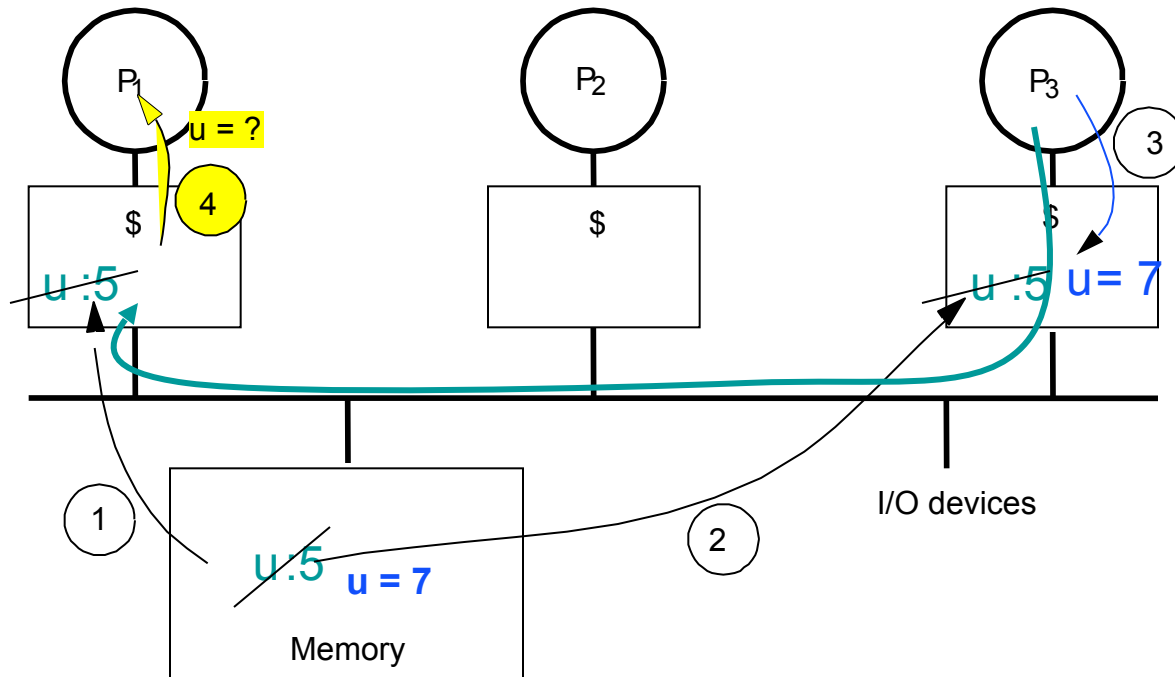


# Cache Coherence Example



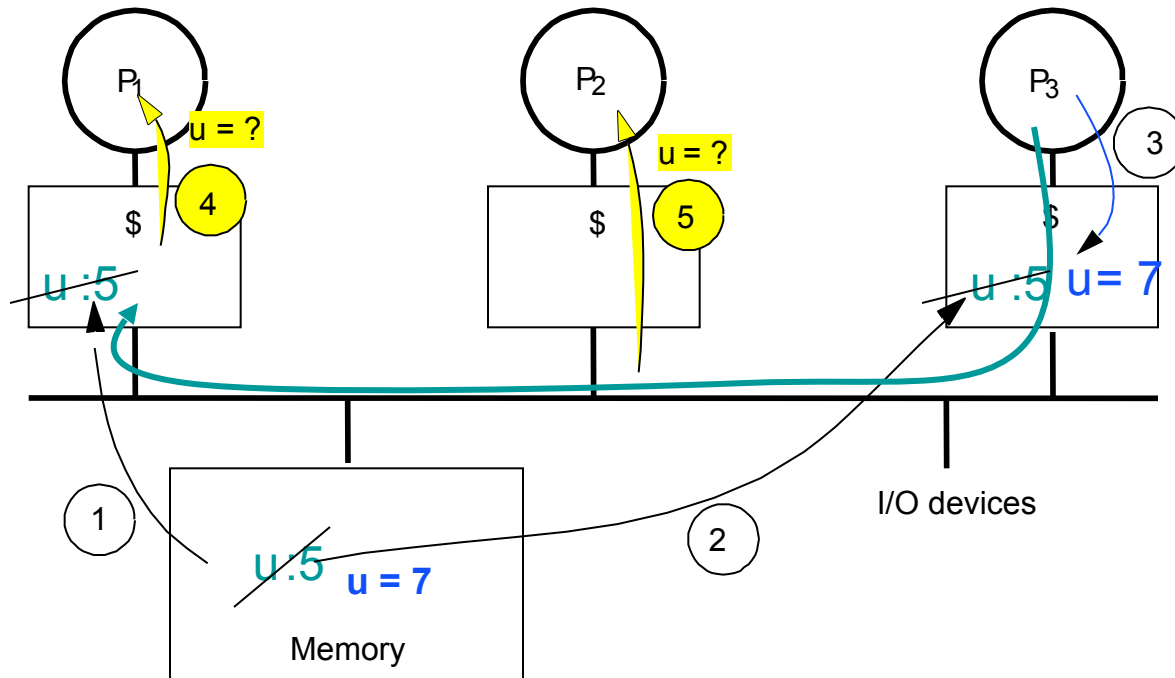
- Must invalidate before step 3

# Cache Coherence Example



- Must invalidate before step 3

# Cache Coherence Example



- Must invalidate before step 3

# Coherence Invariants

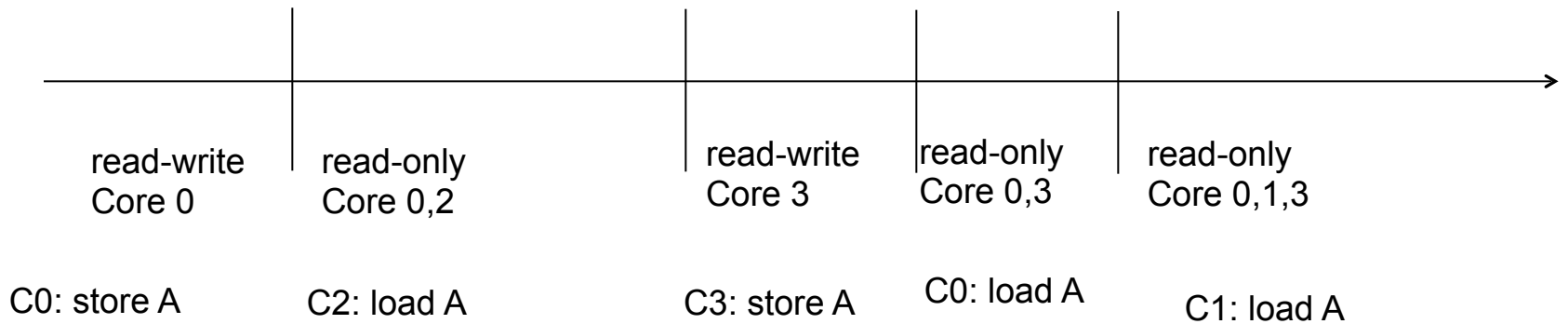
One definition of coherence says these conditions hold:

1. **Single-Writer, Multiple-Reader (SWMR) Invariant.** For any memory location A, at any given (logical) time, there exists only a single core that may write to A (and also read it) or some number of cores that may only read A.
2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.



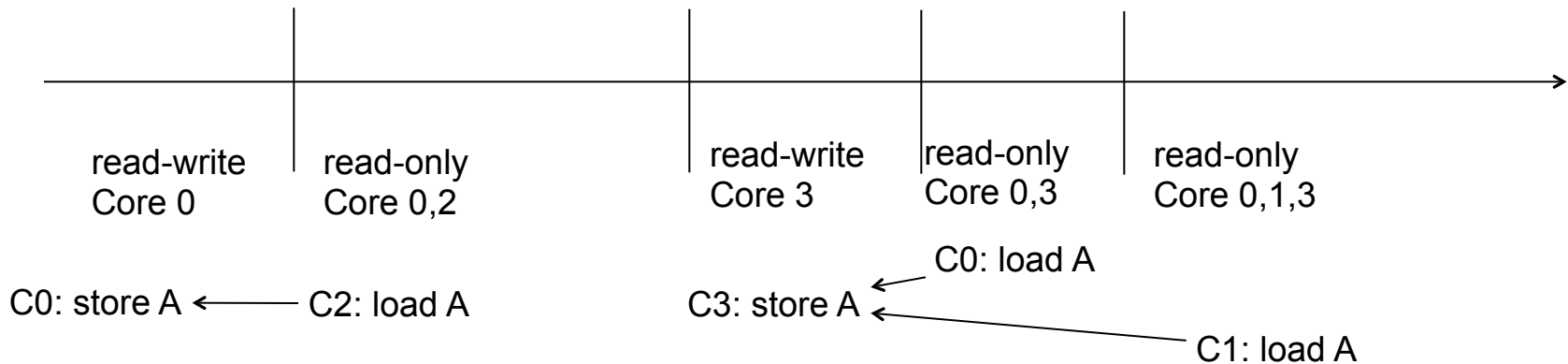
# SWMR Invariant

Lifetime of location A might look like:



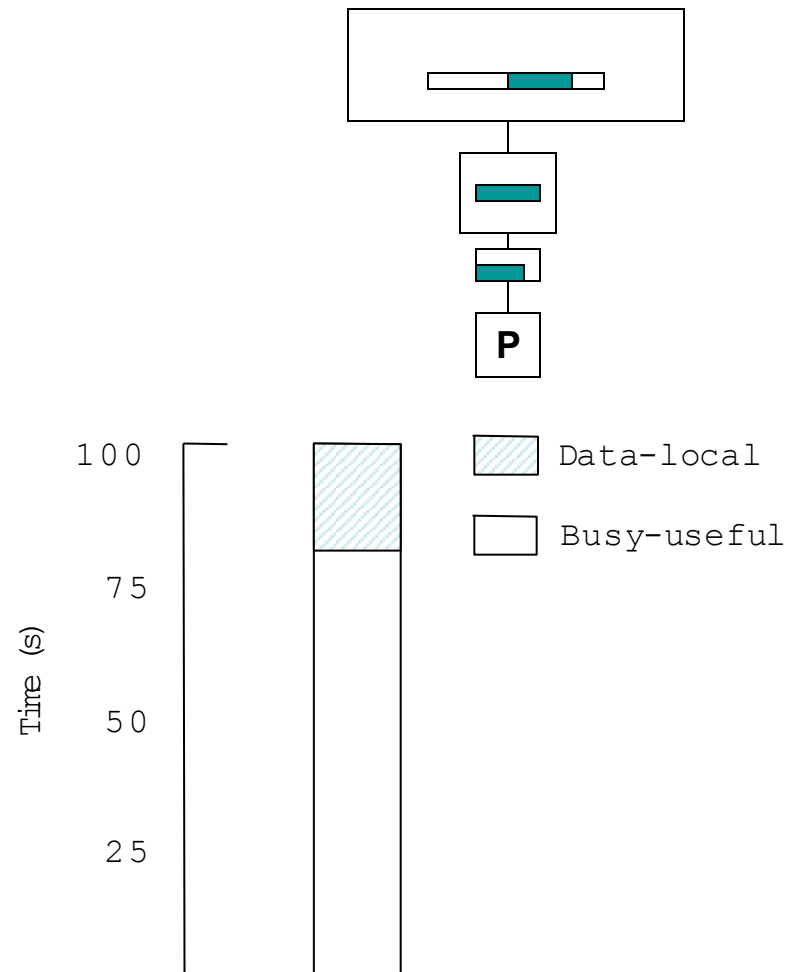
# Data-Value Invariant

Load should get value from last store



# Single Core Performance

- Performance depends heavily on memory hierarchy
- Managed by hardware
- Time spent by a program
  - $\text{Timeprog} = \text{Busy} + \text{Data Access}$
- Data access time can be reduced by:
  - Optimizing machine
    - bigger caches, lower latency...
  - Optimizing program
    - temporal and spatial locality



# Multicore Performance

