# Week 2: Parallel Hardware, Shared Memory Systems and Low-Level Synchronization Primitives

MPCS 52060: Parallel Programming

University of Chicago

# Video 1.1: Parallel Hardware and Multicore Architecture

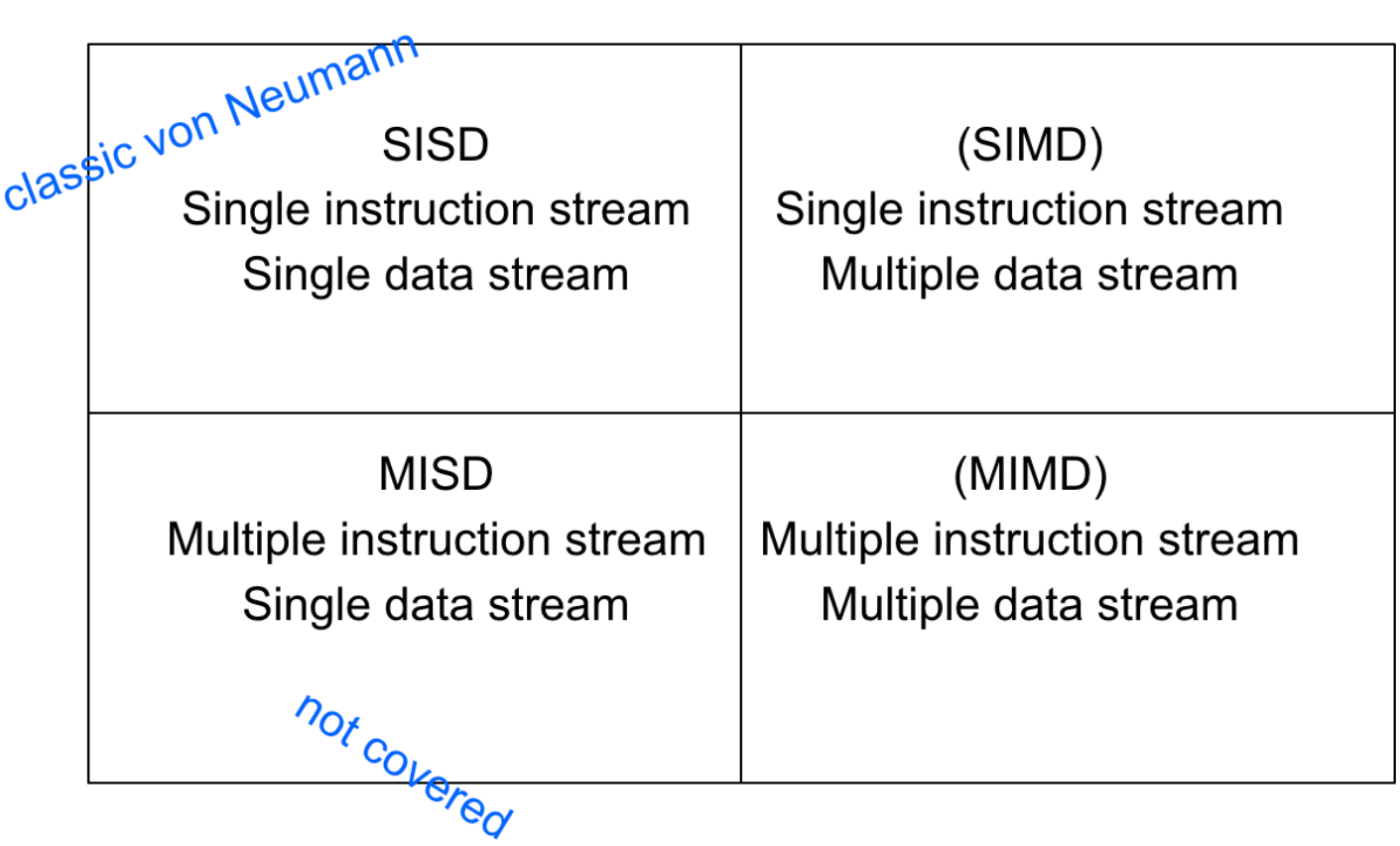
# Video Outline

1. Different classifications of Parallel Hardware
2. Parallelism of Multicore Architecture

# Parallel Hardware

# Flynn's Taxonomy

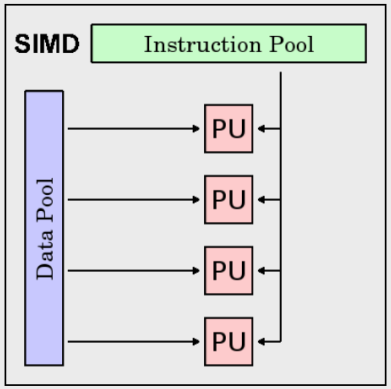
- Many different ways to classify parallel computers.
- One widely used classification is known as **Flynn's taxonomy**:
  - Architectures are classified based on two independent dimensions: *Instruction Stream* and *Data Stream*
  - Each Dimension has two possible states: *Single* and *Multiple*

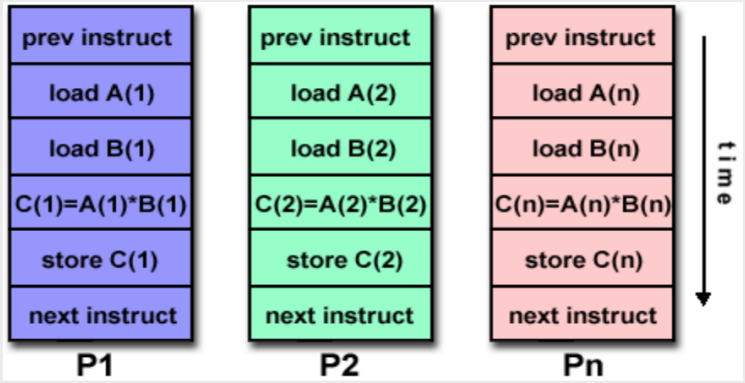


# Single Instruction, Multiple Data (SIMD)

- **Single Instruction**: All processing units execute the same instruction at any given clock cycle
- **Multiple Data**: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Called **data parallelism**.
- Synchronous (lockstep) and deterministic execution
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

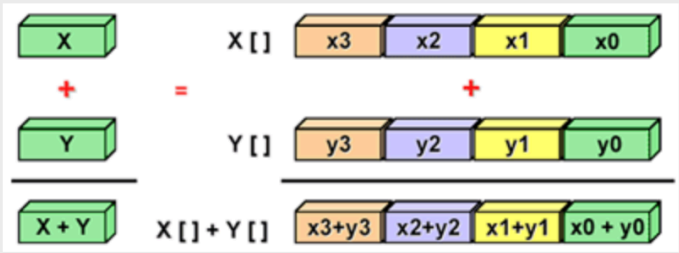
# Single Instruction, Multiple Data (SIMD)



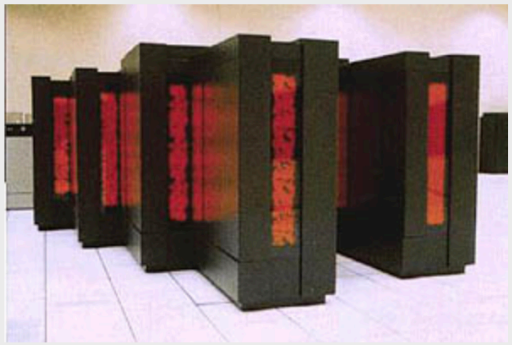

ILLIAC IV

MasPar

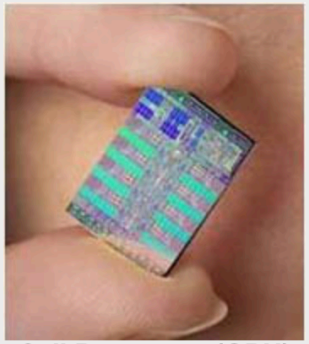



Cray X-MP

Cray Y-MP



Thinking Machines CM-2



Cell Processor (GPU)

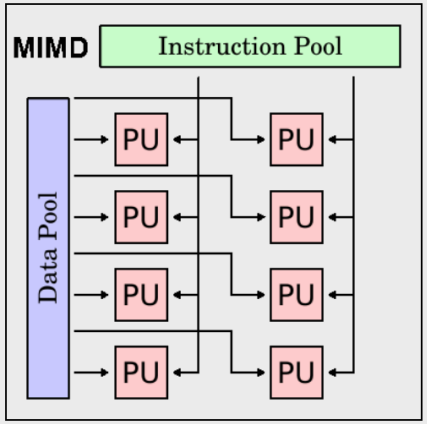
# SIMD drawbacks

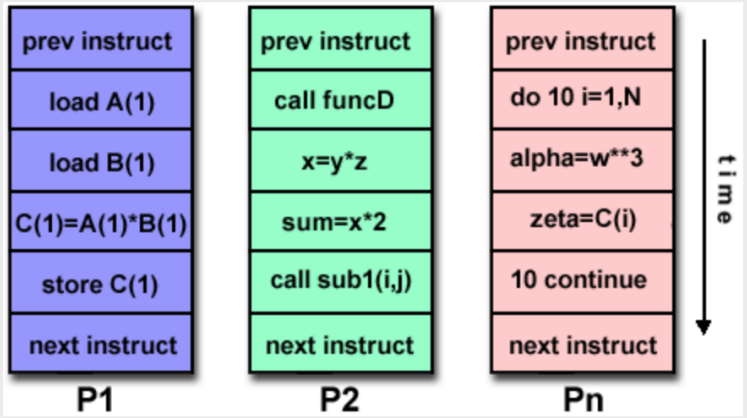
- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.

# Multiple Instruction, Multiple Data (MIMD)

- **Multiple Instruction**: Every processor may be working with a different data stream
- **Multiple Data**: Every processor may be working with a different data stream
- Sometime called **task parallelism**.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Most common type of parallel computer
- Examples: most current supercomputers, networked parallel computer clusters and ”grids”, multi-processor SMP computers, multi-core PCs.
- **Note:** many MIMD architectures also include SIMD execution sub-components

# Multiple Instruction, Multiple Data (MIMD)

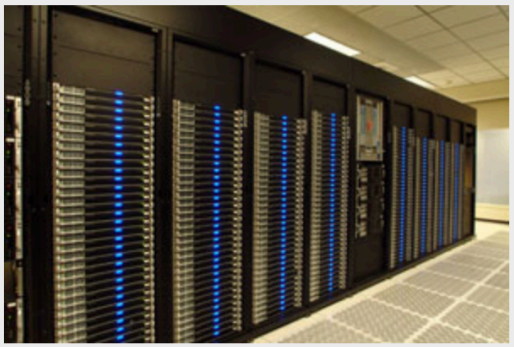



IBM POWER5

HP/Compaq Alphaserver

Intel IA32



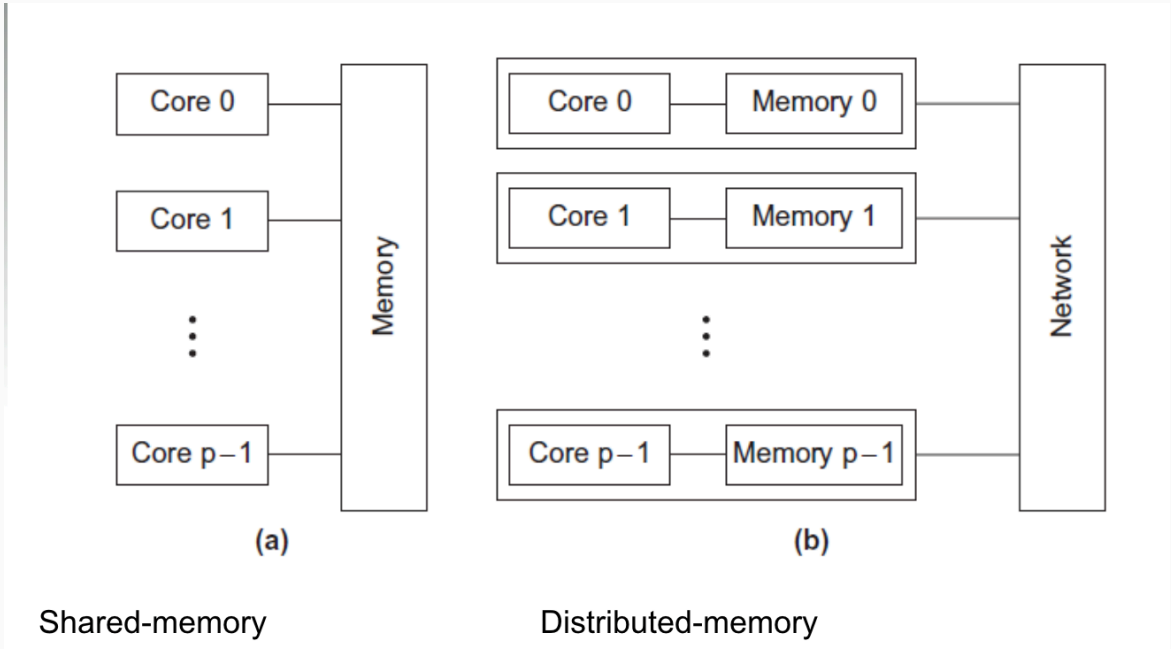
AMD Opteron

Cray XT3

IBM BG/L

# Types of MIMD Systems

- Shared-memory (**The focus of this course**)
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory (**The focus of Distributed Systems and HPC courses**)
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.



# Parallelism in Multi-Core Architectures

# Review: Multicore Processors

As stated in last lecture, the design trend for parallel architectures have moved multi core chips:

- Consists of multiple hardware processors(CPUs), each which executes a sequential program.

For multicore architectures, the basic unit of time is a cycle:

- The time it takes a processor to fetch and execute a single instruction
- As technology advances, cycle times change
  - 1980: 10 million cycles/sec
  - 2005: A 3 GHz processor does 3 billion cycles/sec
  - Some instructions take one cycle, and some may take hundreds.

# Multi-Core Architecture

Modern hardware architectures have a multi-core architecture, where both serial and parallel programs have benefited from *internal parallelism*, which is parallelism being down at the hardware level:

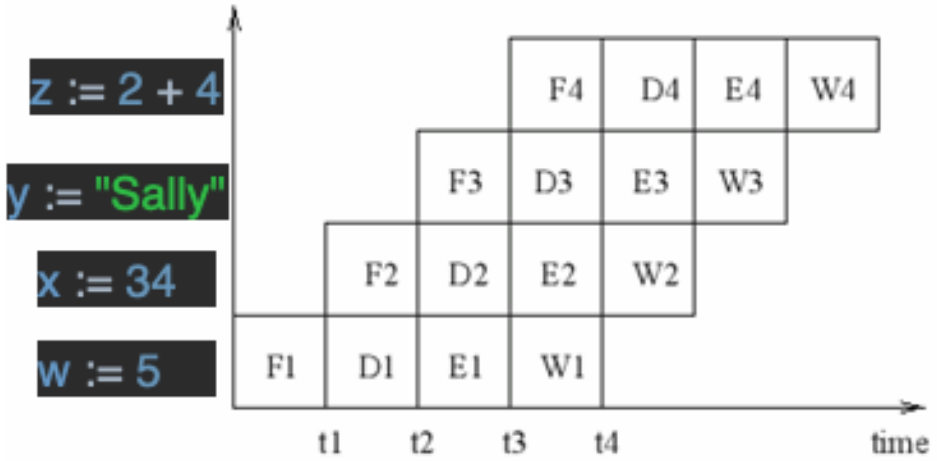
- **Bit level parallelism**:
  - Word size by the processors increased stepwise from 4 to 32-bit.
  - Increasing the word size lead to less processing time to perform 32-bit operations.
  - Trend has stopped with the adoption of 64-bit operations, which gives sufficient accuracy for floating point numbers and covers a large address space.
- **Pipelining parallelism**:
  - Pipelining at the instruction level is an overlapping of the execution of multiple instructions.
  - Execution of each instruction is partitioned into several steps which are performed by dedicated hardware units (pipeline states) one after another.

# Internal Parallelism: Pipelining

Simple example of pipeline stages:

- fetch: fectch the next instruction to be executed from memory
- decode: decode the instruction fected in step (1).
- execute: load the operands specified and execute the instruction
- write back: write the result into the target location.

```
func main() {

    w := 5
    x := 34
    y := "Sally"
    z := 2 + 4
    //...
}
```




**Main benefit**: Different pipeline stages can operate in parallel, if there are no control or data dependencies between the instructions to be executed.

# Internal Parallelism (cont.)

- **Multiple functional units parallelism**:
  - Many processors has multiple, independent functional units like ALUS (arithemtic logical units), FPUs(floating point unit), load/store units, or branch units.
  - With these units, different independent instructions can be executed in parallel by different functional un its.
  - Processors can execute instructions speculatively before branches or data have been computed.
  - Processors that allows this functional unit parallelism are known as superscalar processsors.

All three internal parallelism techniques discussed are handled by either **compilers** or **interpreters** and make sequential and parallel programs more efficient and have better performance. As stated in last lecture, we are limited by how much performance we can achieve with just this internal parallelism.

# 1.2: Shared Memory Systems

# Video Outline

1. Overview of Shared Memory Systems
2. Programming of Shared Memory Systems
3. Components and variants of Shared Memory Systems

# Computers with Shared Memory Architecture

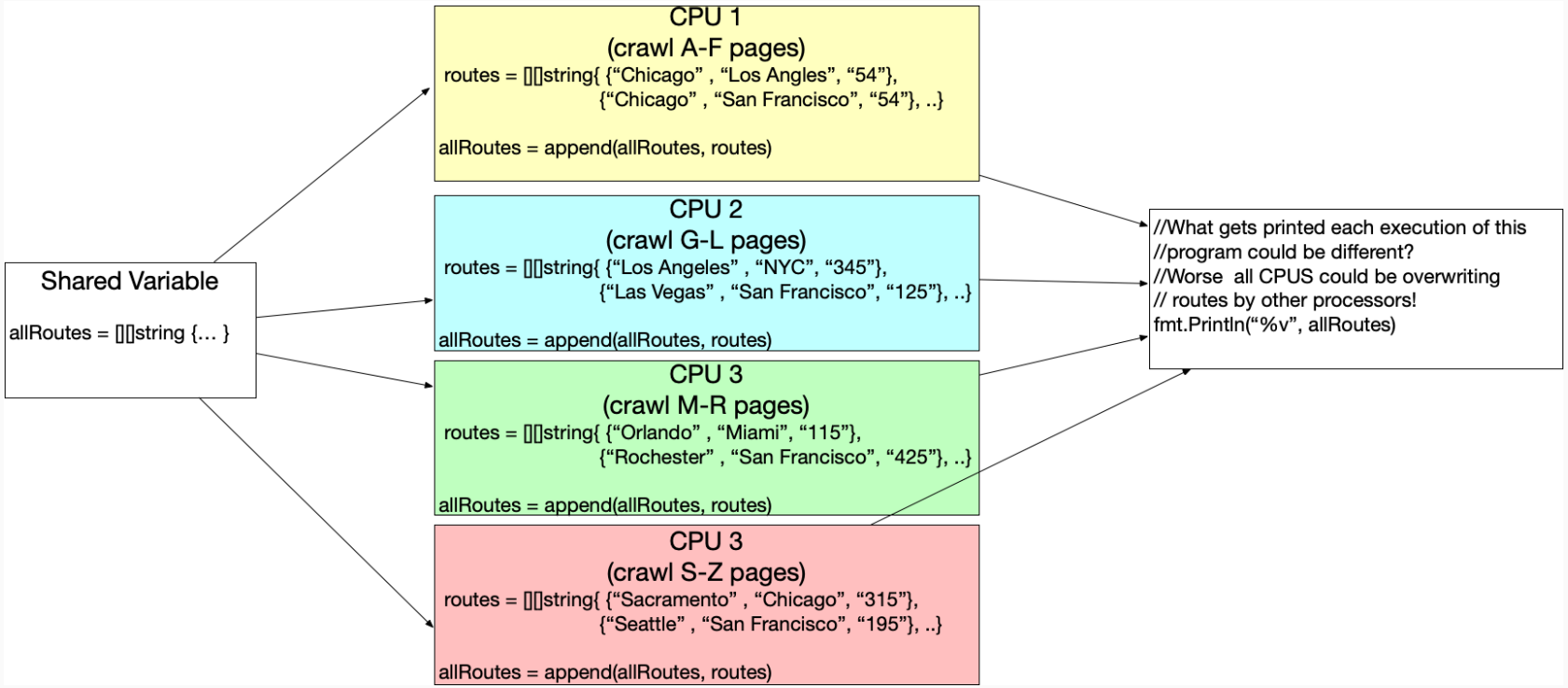
Computers with physically shared memory are also called shared memory machines (SMMS). The components of the architecture consist of the following:

- Number of processors or cores
- A shared physical memory (also known as global memory)
- Interconnection network to connect the processors with the memory.
- Shared memory can be implemented as a set of memory modules (e.g., caches, physical memory, etc.)

# Parallel Programming on SMMs

- SMMs use **shared variables** which can be accessed by all processors.
- Communication and cooperation between the processors is organized by writing and reading shared variables that are stored in global memory.
- Accessing shared variables concurrently by several processors should be avoided since **race conditions**:
  - A situation in which the program produces unexpected results
  - Hard to reproduce and diagnose because they can be appear in frequently.
  - Only present sometimes under heavy load or when using certain compilers, platforms, architectures

# Race condition




**Fix:** Need synchronization constructs to ensure deterministic results:

- A program is deterministic if it always produce the same output from a given starting condition or initial state.
- Focus of the next few lectures.

# Parallel Programming on SMMs (cont.)

Parallel programs for SMMs are often based on the execution of threads:

- An instance of a computer program that is being executed. Where a processor is a hardware device, a thread is a *software construct*.

A program can actually have multiple threads associated with it. The *main thread* (or *"heavy"* thread) is known as an operating system process.

Components of a process:

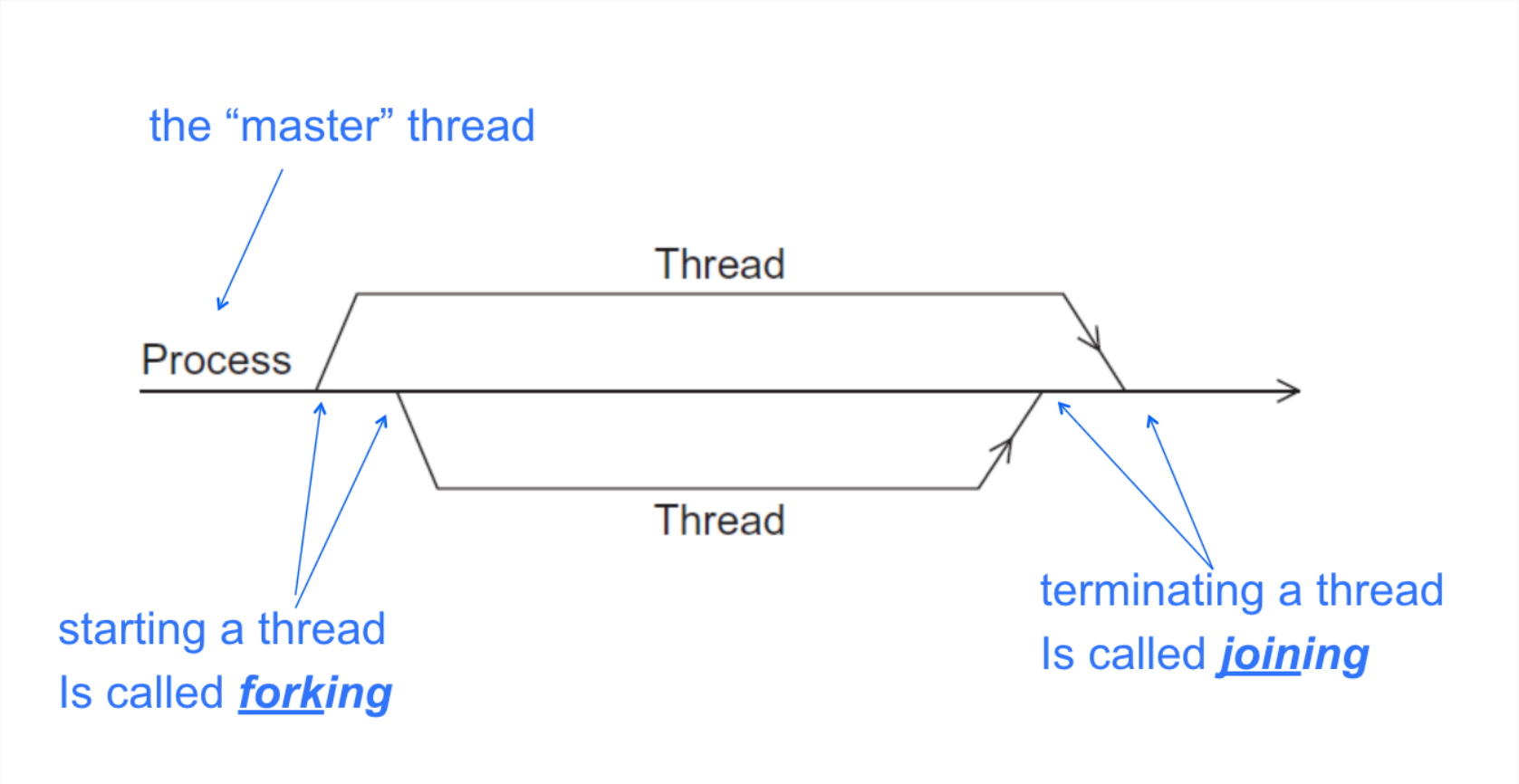
- The executable machine language program
- A block of memory
- Descriptors of resources the OS has allocated to the process
- Security information
- Information about the state of the process

# Processes and Threads

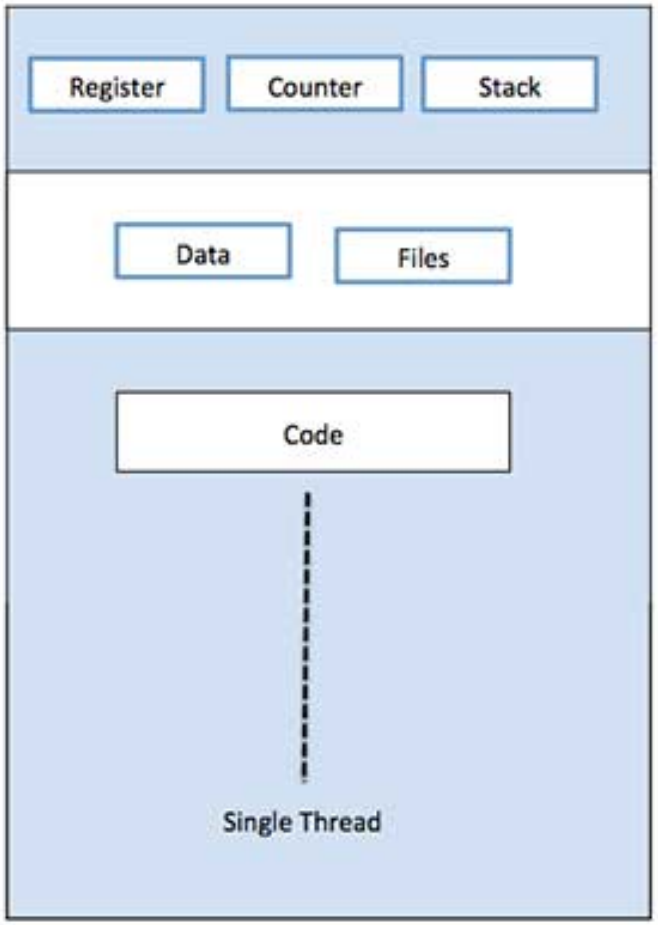
A process can contain a number of *"light-weight"* threads that can:

- Allow programmers to decompose the computations of their application into several parts (known as tasks) that can be executed in parallel.
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.
- Have their own stack where they can store local variables, function calls, etc.
- Some languages and operating systems provide the notion of thread-local storage, where threads can store and retrieve values independent of other threads.

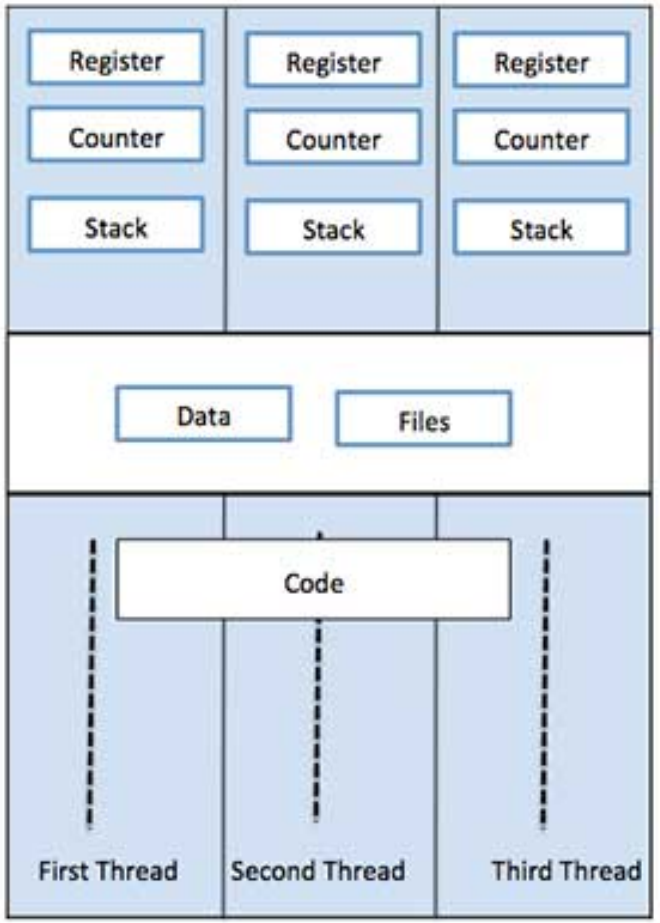
# Example: A process and two threads



# Example: A process and three threads




Single Process P with single thread




Single Process P with three threads [1]

[1] Operating System - Multi-Threading

# Processes and Processors

Gives the illusion that a single processor system is running multiple programs simultaneously.

- Each process takes turns running (i.e., time slice).
- A processor can run a process for a while and then set it aside and run another process (i.e., context switch)
- After its time is up, it waits (i.e., (blocks) until it has a turn again.

Processor may set aside or *descheduled* a process for a number of reasons:

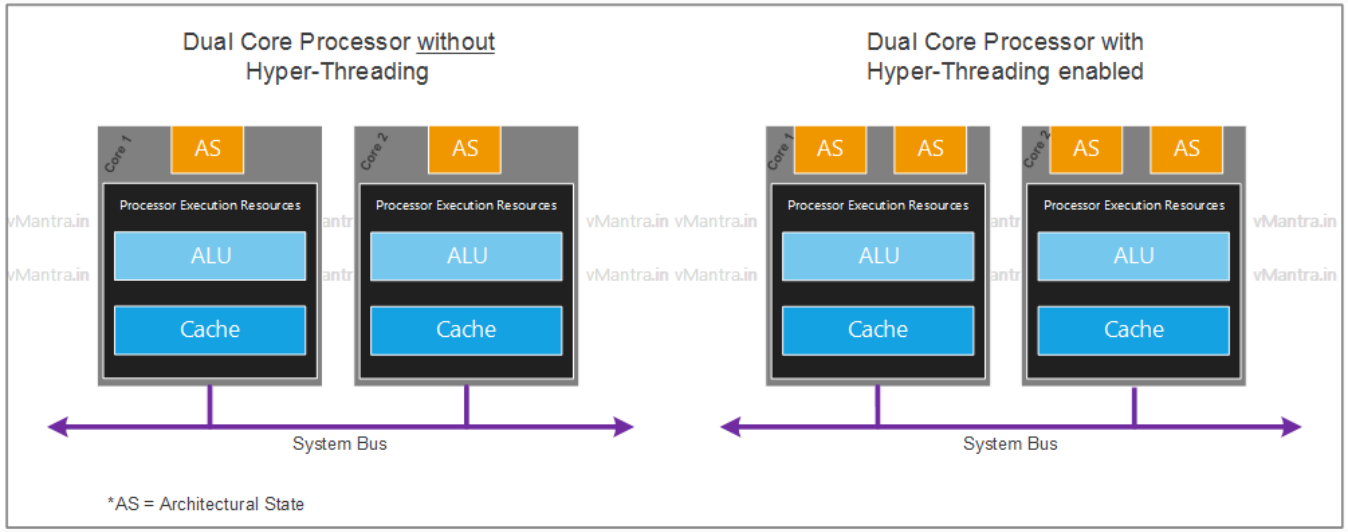
- A memory request that will take some time to satisfy
- A process has run long enough (i.e., reached an end to its time slice.). Thus, it's time for another process to begin its time slice.

**Note:** When a process is descheduled, it may resume execution on another processor.

# Aside: Simultaneous Multithreading (aka HyperThreading)

Most modern multicore architectures have Simultaneous multithreading(SMT):

- Use several threads and to schedule executable instructions from different threads in the same cycle if necessary.
- It helps increase the usage of functional units of a processor more effectively.
- Hardware support for SMT is based on the replication of the chip area used to store the processor state.




[2]

[2] http://vmantra.in/hyper-threading/

# Aside: Simultaneous Multithreading (aka HyperThreading)

- The processor appears to the operating system and user programs as a set of **logical processors** to which processors or threads can be assigned for execution.
- Processes or threads can come from a single or several user programs.
- Number of replications of the processor state determines the number of logical processors.
- These logical processors are also known as hardware threads

# Interconnect

The interconnect is the medium by which processors communicate with main memory and other processors (also known as the *system bus*).

We will only look at systems where their interconnect is built off shared-memory architectures:

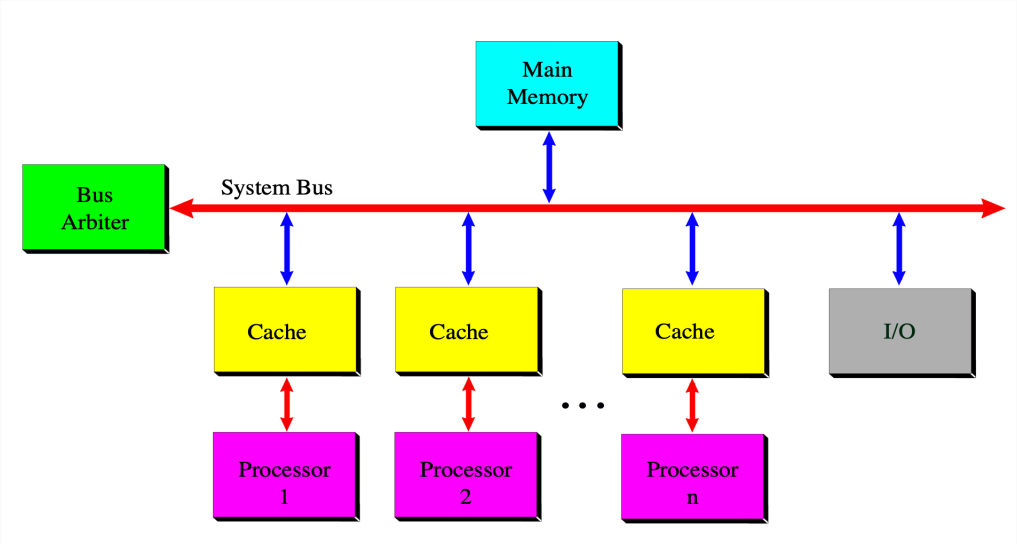
- Essentially their are two heavily-used shared-memory interconnect architectures: Symmetric Multiprocessing(SMP) and Nonuniform memory access(NUMA).

# Symmetric Multiprocessing

- A special variant of a shared memory system.
- All processors can access the shared main memory at the same speed.
- Controlled by a single operating system instance that treats all processors equally (no processor is reserved for a specific purpose).
- SMP systems are *tightly coupled* multiprocessor systems,
  - All processors can execute different programs (and with different data) in parallel.
  - All processors share common resources (e.g., memory, I/O device, interrupt system) all on the same system bus.

# Symmetric Multiprocessing (cont.)

- Processors and memory are linked by a *system bus*(a broadcast medium that acts like a tiny Ethernet).
- Processors and memory have *bus controllers* units in charge of sending and listening for messages broadcast on the bus (listening is sometimes called snooping).
- **Advantage**: Most common interconnect architecture used today because its easy to build.
- **Disadvantage**: Not scalable to large number of processors because the bus becomes overloaded.



# Interconnect cont.

- Interconnect is a finite resource shared among processors
- Performance in SMP systems is limited by memory bus bandwidth
- Processors can be delayed if others are consuming too much of the interconnect's bandwidth
- SMP configurations do not scale well past 64 processors

# Processors and Memory in Shared Memory Systems

# Processors & Memory

On architectural principle drives everything else: *processors and main memory are far apart.*

- Takes a long time to read a value from memory
- Takes a long time for a processor to write a value to memory
- Takes a longer time for the processor to verify that the value written is installed in memory.
- The relative cost of instructions such as memory access changes slowly when expressed in terms of cycles.
- **Analogy**: Accessing memory is more like mailing a letter than making a phone call.

Memory access time has a large influence on program performance. The objective of architecture trends over the years have been to reduce memory access latency:

- The total time that elapses until a memory access operation has been completely terminated.

# Processors & Memory (cont.)

To help alleviate these memory latency issues, memory inside modern computers is actually a *hierarchy* of components that store data:

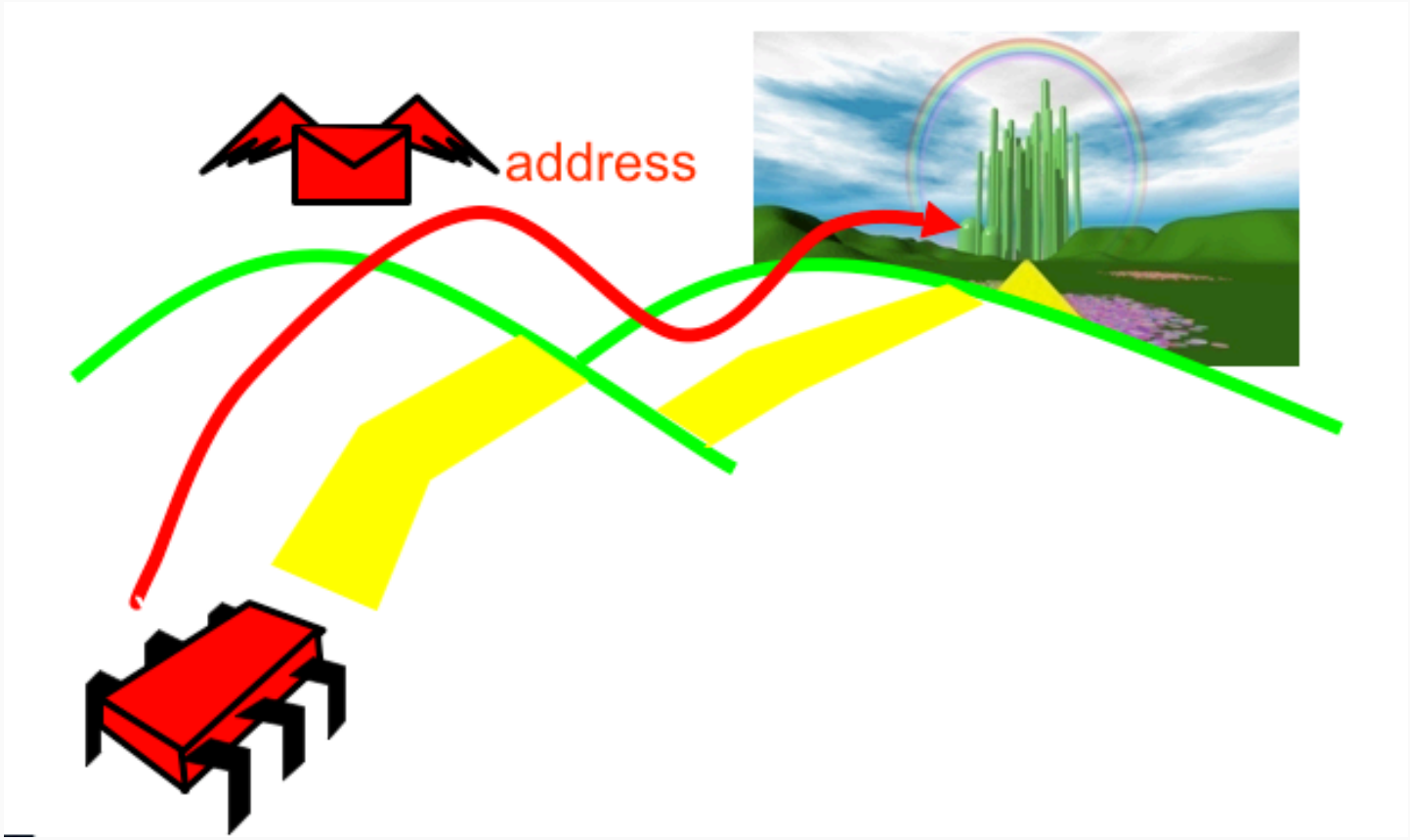
- Ranges from one or more levels of small, fast caches to relatively slow main memory.
- **Understanding how these levels interact is essential to understanding the actual performance of many concurrent algorithms**.

# Main Memory

- Large array of words(i.e., memory blocks/locations), indexed by an address.
- Both words and addresses are typically either 32 or 64 bits.

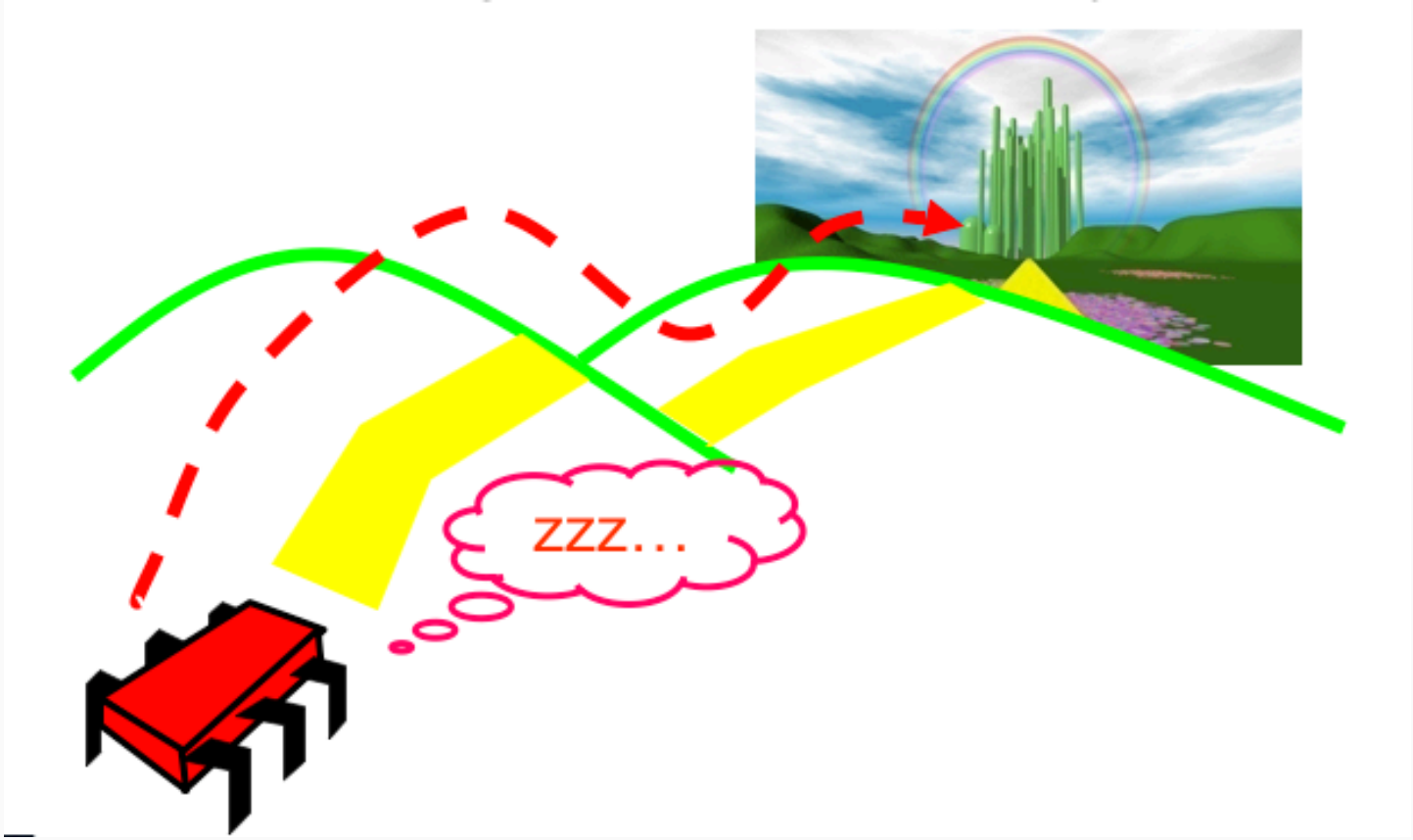
# Reading from Memory

Processor sends an address to main memory that it wishes to retrieve data from



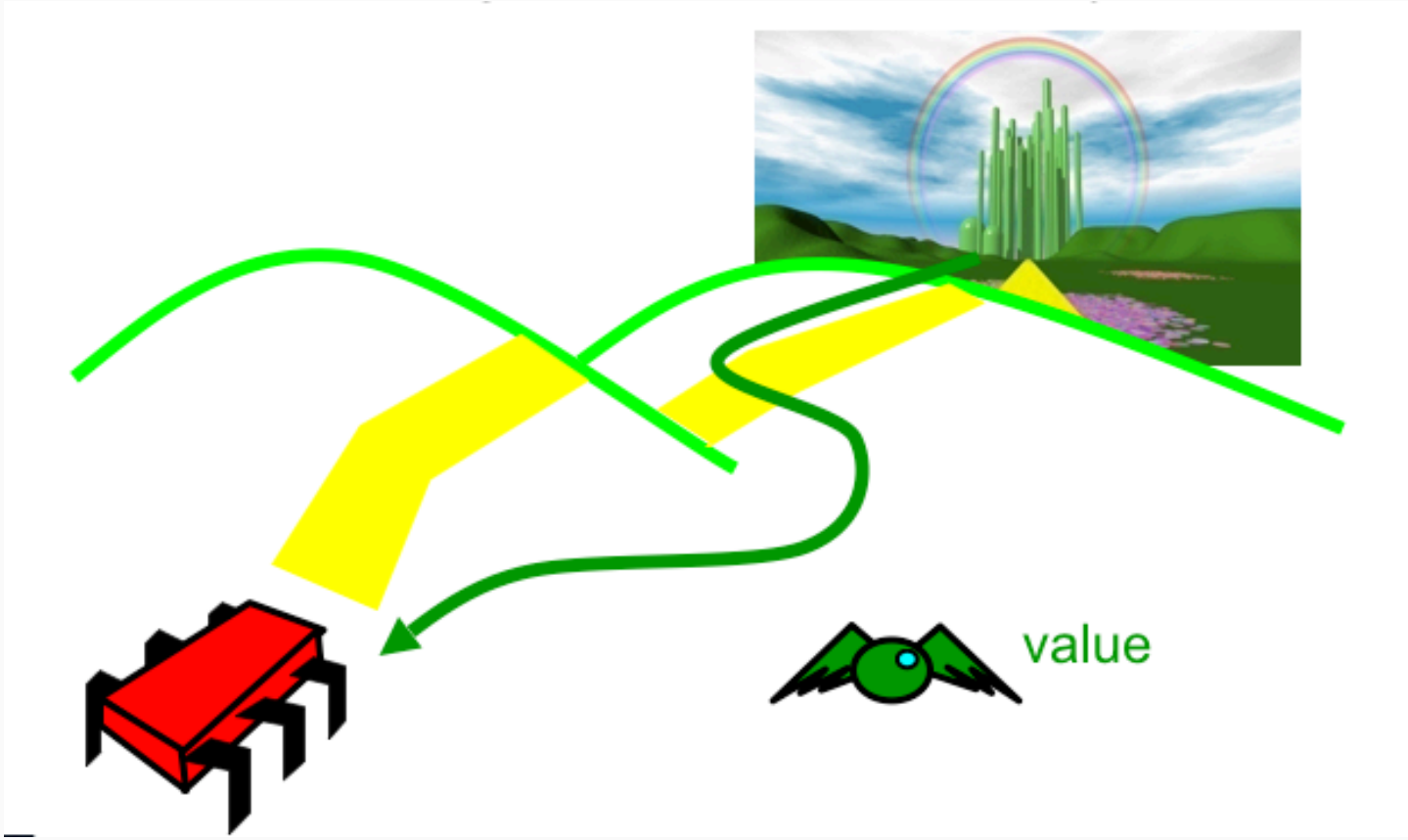
# Reading from Memory

Processor waits until main memory sends data back to it.



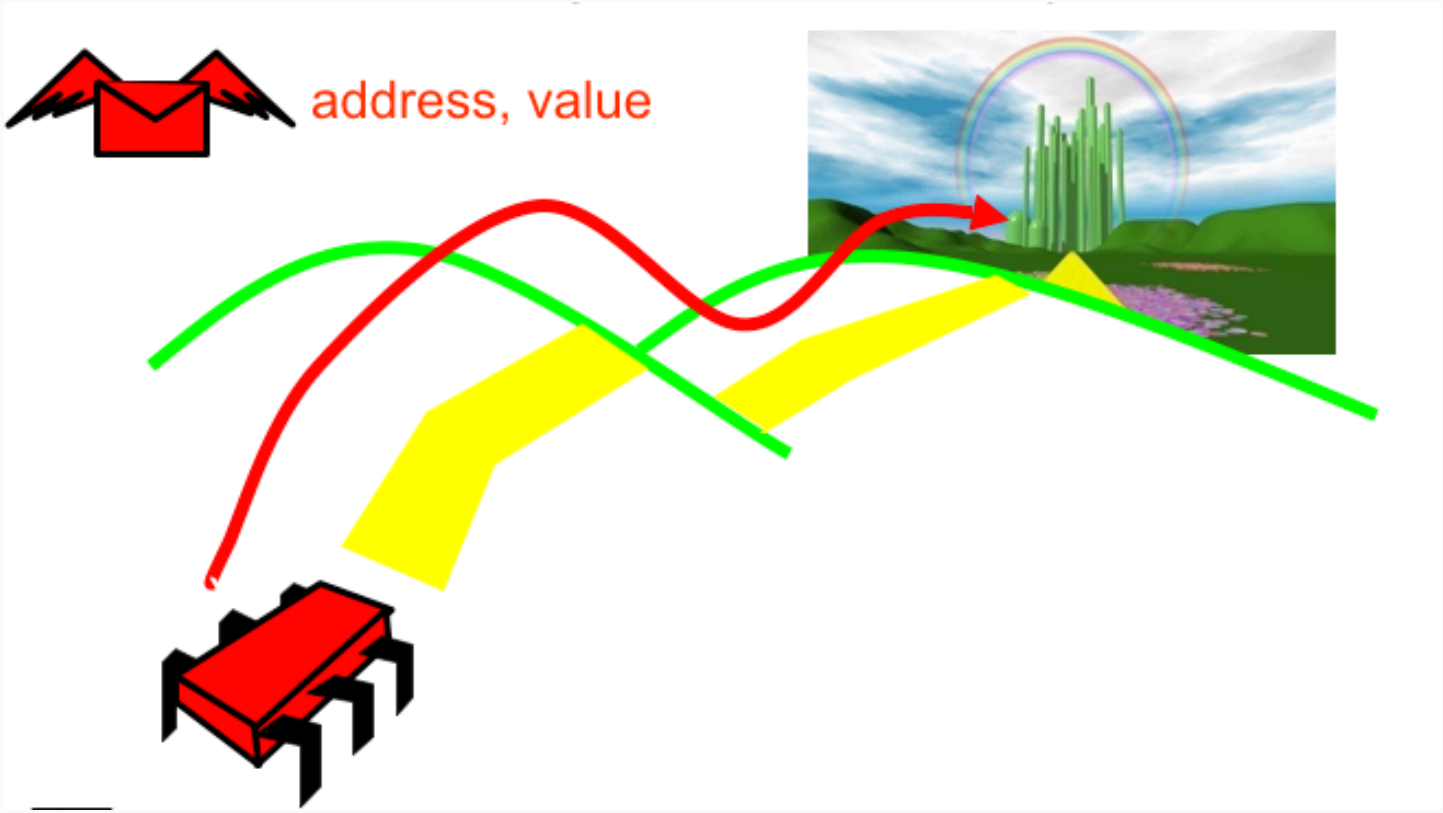
# Reading from Memory

Processor finally receives the data from main memory and can continue executing.



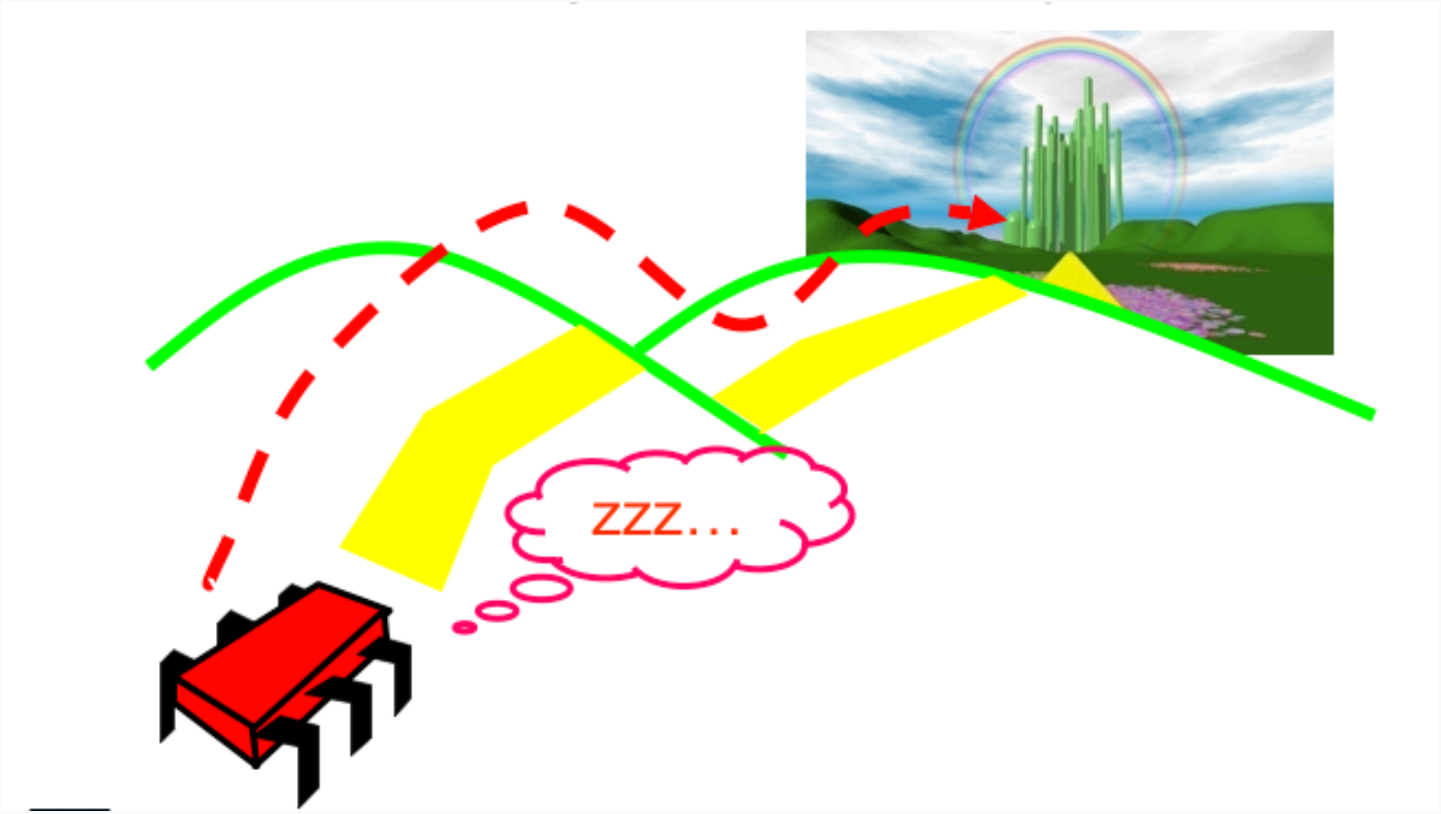
# Writing to Memory

Processor sends an address and data to write at the address to main memory.



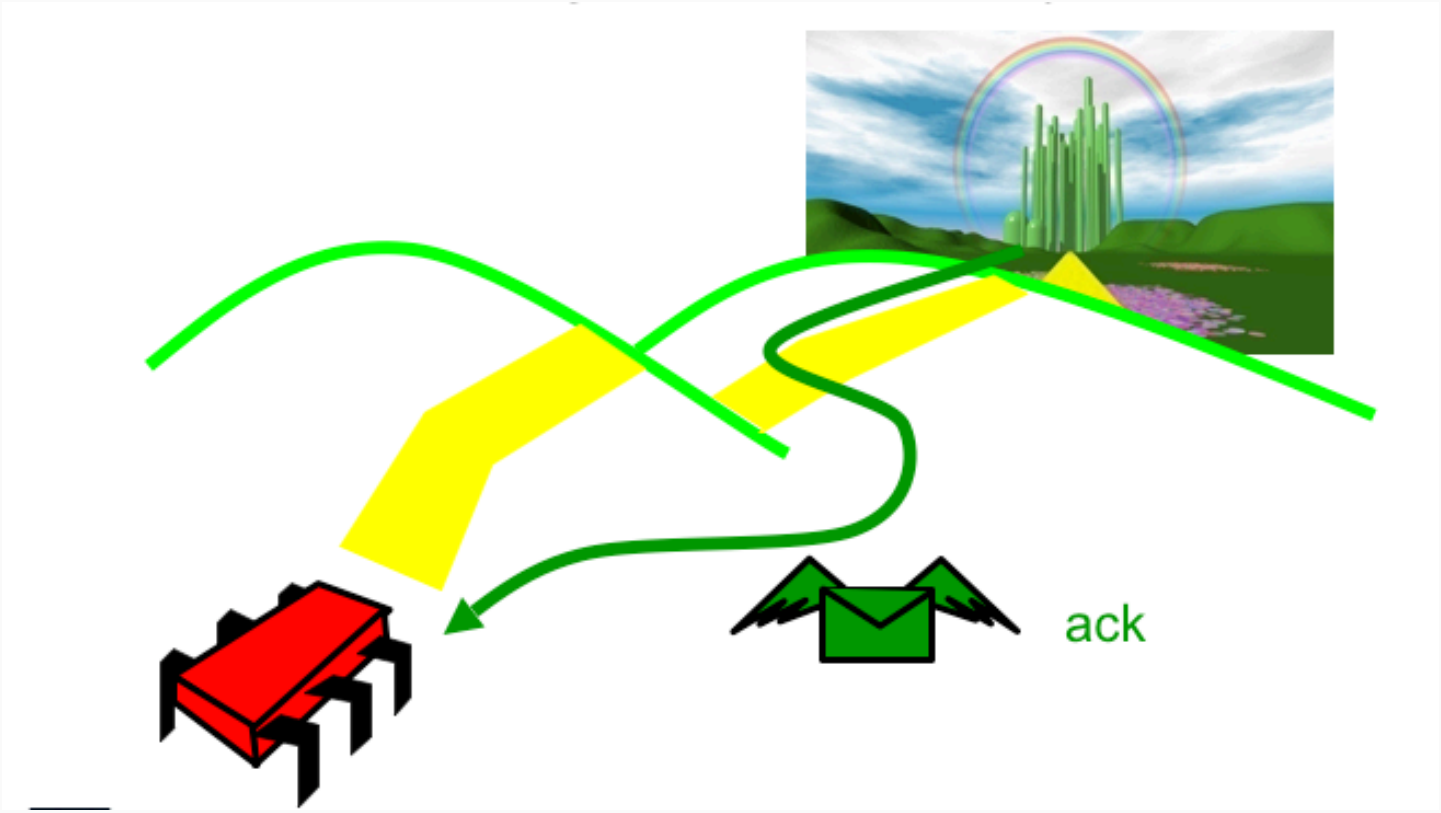
# Writing to Memory

Processor waits for acknowledgement that main memory wrote the data



# Writing to Memory

Processor receives the acknowledgement and can continue executing



# 1.3: Caches and Programming on Shared Memory Syesms

# Video Outline

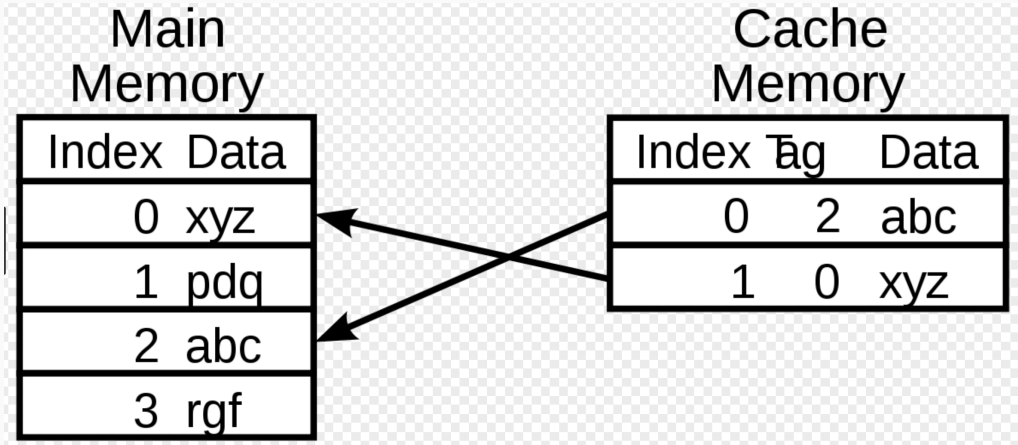
1. Finish discussion on memory hierarchy of SMMs
2. Programming of Shared Memory Systems
3. Low-Level synchronization primitives

# Caches

On modern architectures, processors can waste hundreds of CPU cycles waiting to access main memory.

We can alleviate problem by using caches:

- A collection of memory locations that can be accessed in less time than some other memory locations.
- A cache is typically located on the same chip as the processors
- Cache line: fixed-size block of data that also contains metadata (e..g, tag, index)
- Cache lines are normally 64 or 128 bytes



# Principle of locality

Caches are effective because most programs display a high degree of locality:

- Accessing one location is followed by an access of a nearby location.
- Types of locality
  - **Spatial locality** – accessing a nearby location.
  - **Temporal locality** – accessing in the near future.

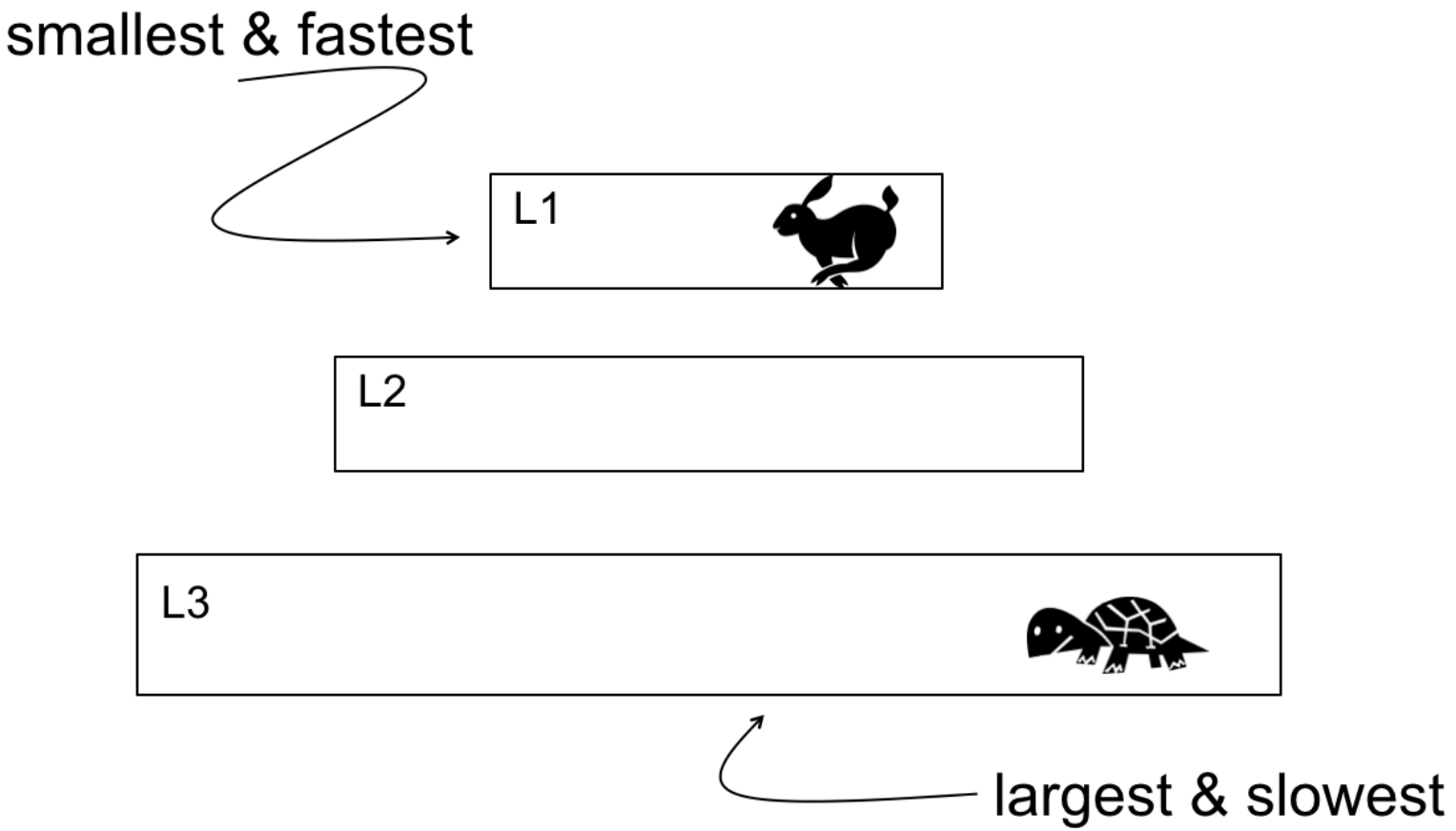
# Cache Levels

Most processors have two levels of caches, called L1, L2, and L3:

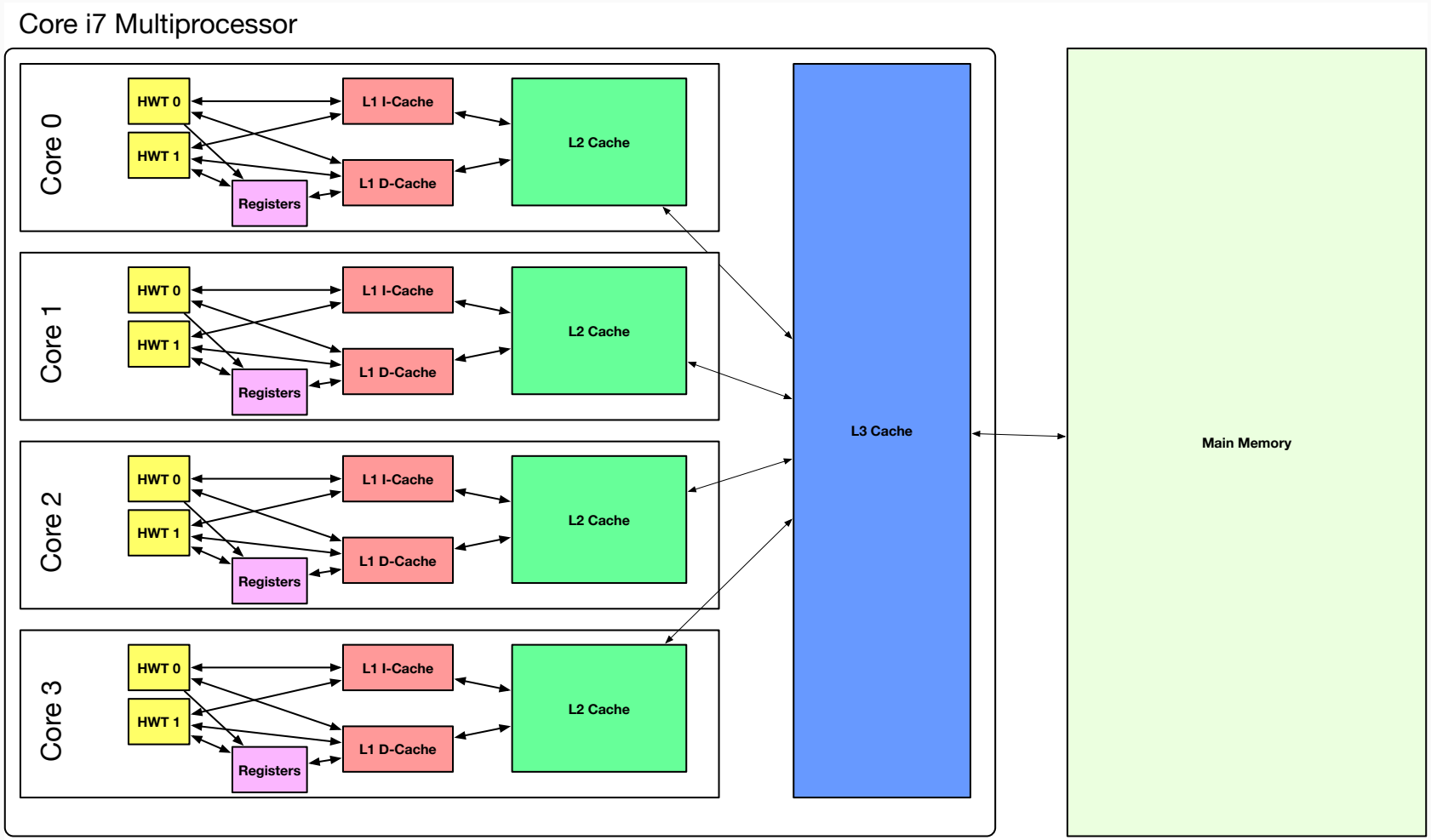
- L1 - typically resides on the same chip as the processor and takes 1-2 cycles to access
- L2 - may reside either on or off-ship, and takes about 10 cycles to access
- L3 - normally is off chip but is accessible within about 30 cycles
- **Note**: These times vary from platform to platform.

All these are significantly faster to access then main memory ( 100s of cycles).

# Levels of Cache



# Core i7-9xx Cache Hierarchy




L1 i-cache and d-cache: 32KB, 8-way, Access: 4 cycles

L2 cache: 256KB, 8-way, Access: 11 cycles

L3 cache: 8MB, 16-way, Access: 30-40 cycles

Cache-Line (Block) size: 64 bytes
(All caches)

# Principle of locality

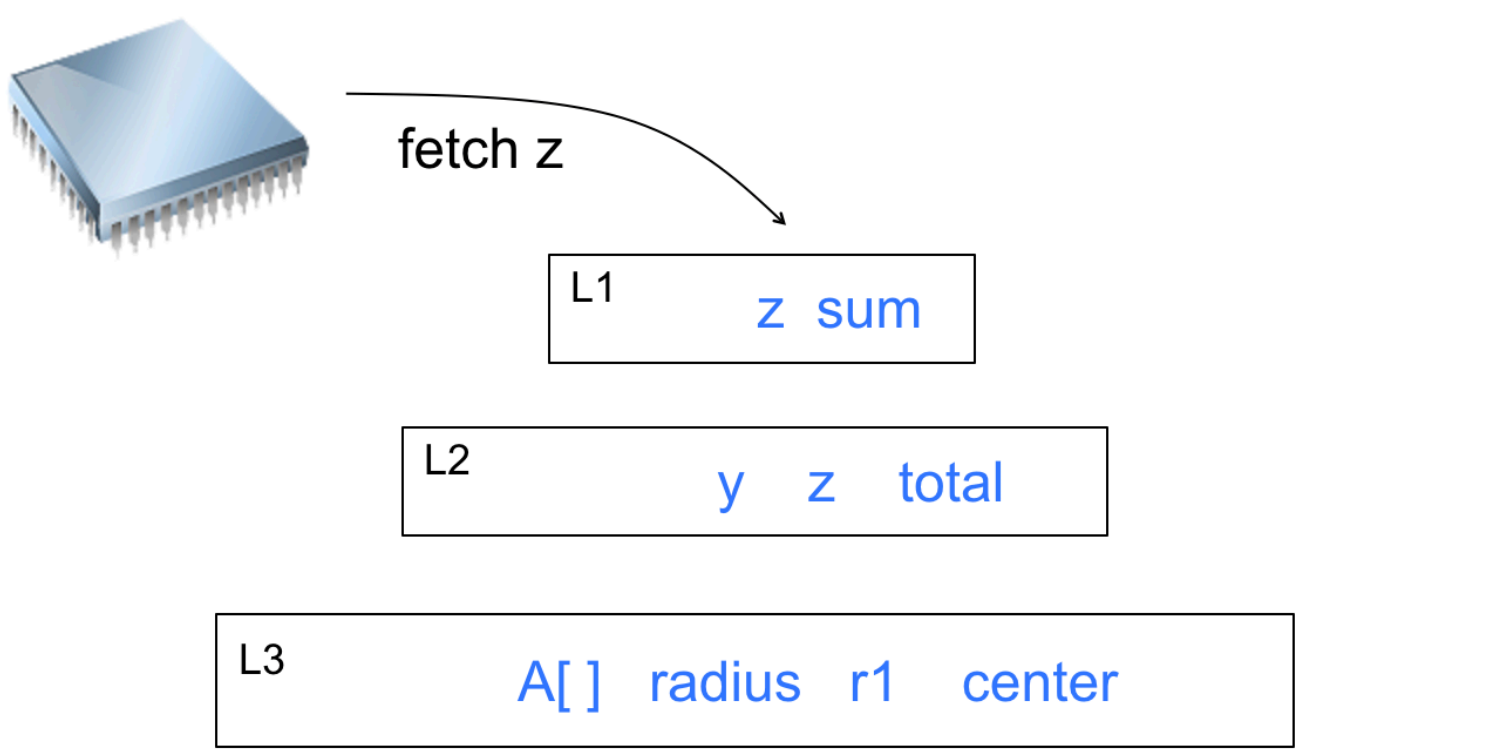
What could the caches levels look like for this program?

```
package main

import "fmt"

func main() {

    var z [1000]int
    var sum int
    for i := 0; i < 1000; i++ {
        sum += z[i]
    }
    fmt.Printf("%v\n", sum)
}
```

# Cache hit

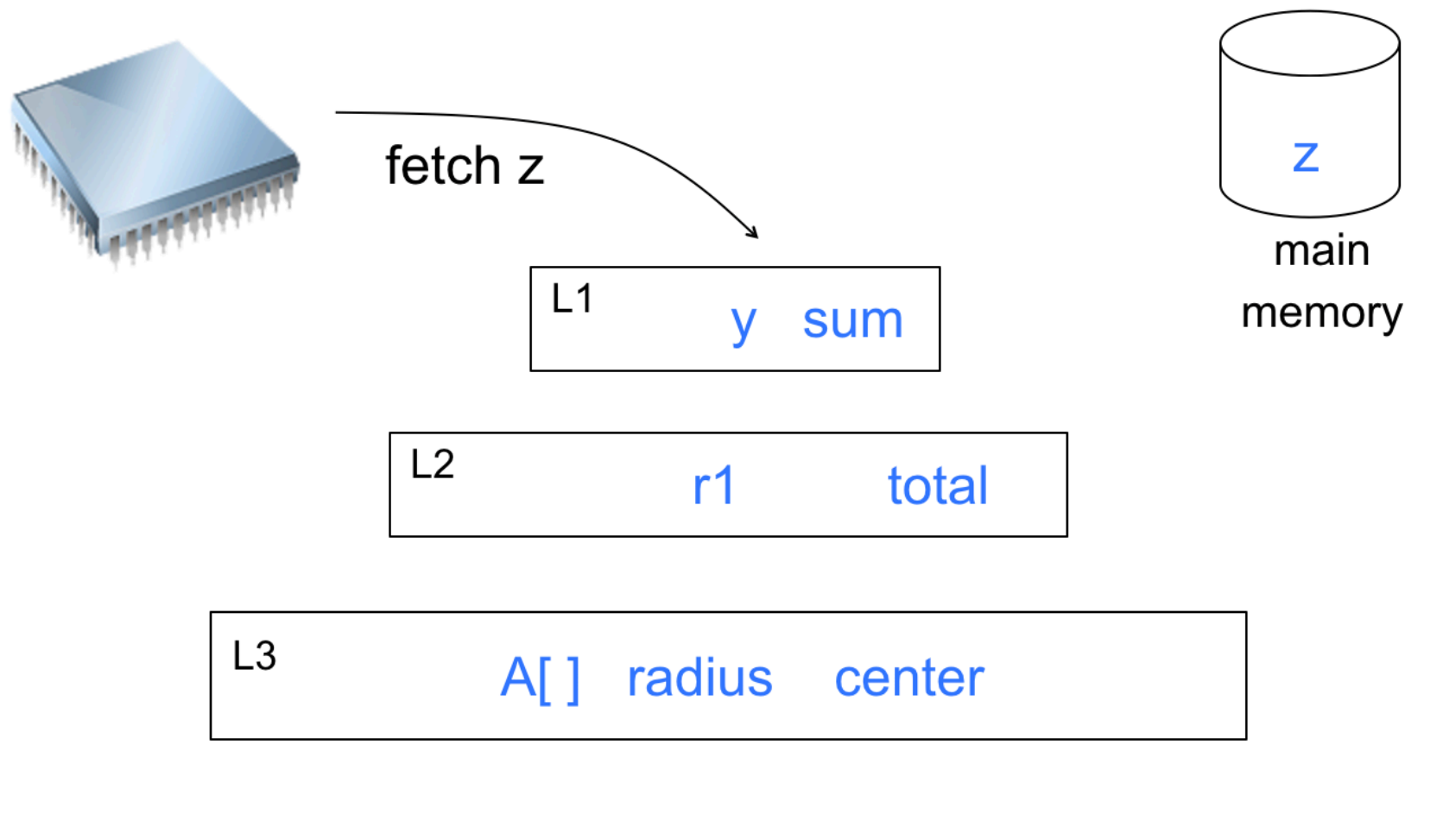
When a processor attempts to read a value from a given memory address, it first checks the cache(s).

We call it a cache hit if the value for the processor is located in one of the cache levels.



# Cache miss

Otherwise, a cache miss is when a value is not in the cache and the process is required to go to main memory.



# When a Cache Becomes Full...

- Caches are expensive to build and therefore significantly smaller than main memory.
- Need to make room for new entry when the cache is full
- By evicting an existing entry:
  - Discarding a entry if it has not been modified
  - Writing it back to main memory if it has

Need a replacement policy(determines which cache line to replace to make room for a new location).

- Usually some kind of least recently used heuristic

# Cache Coherence

- Processor **A** and Processor **B** both cache an address **x**
- Processor **A** writes to **x**
- This operation updates the cache
- How does Processor **B** find out about the update?
- A cache coherence protocols provides a specification on how to keep caches in sync with each other. Many cache coherence protocols in the literature.

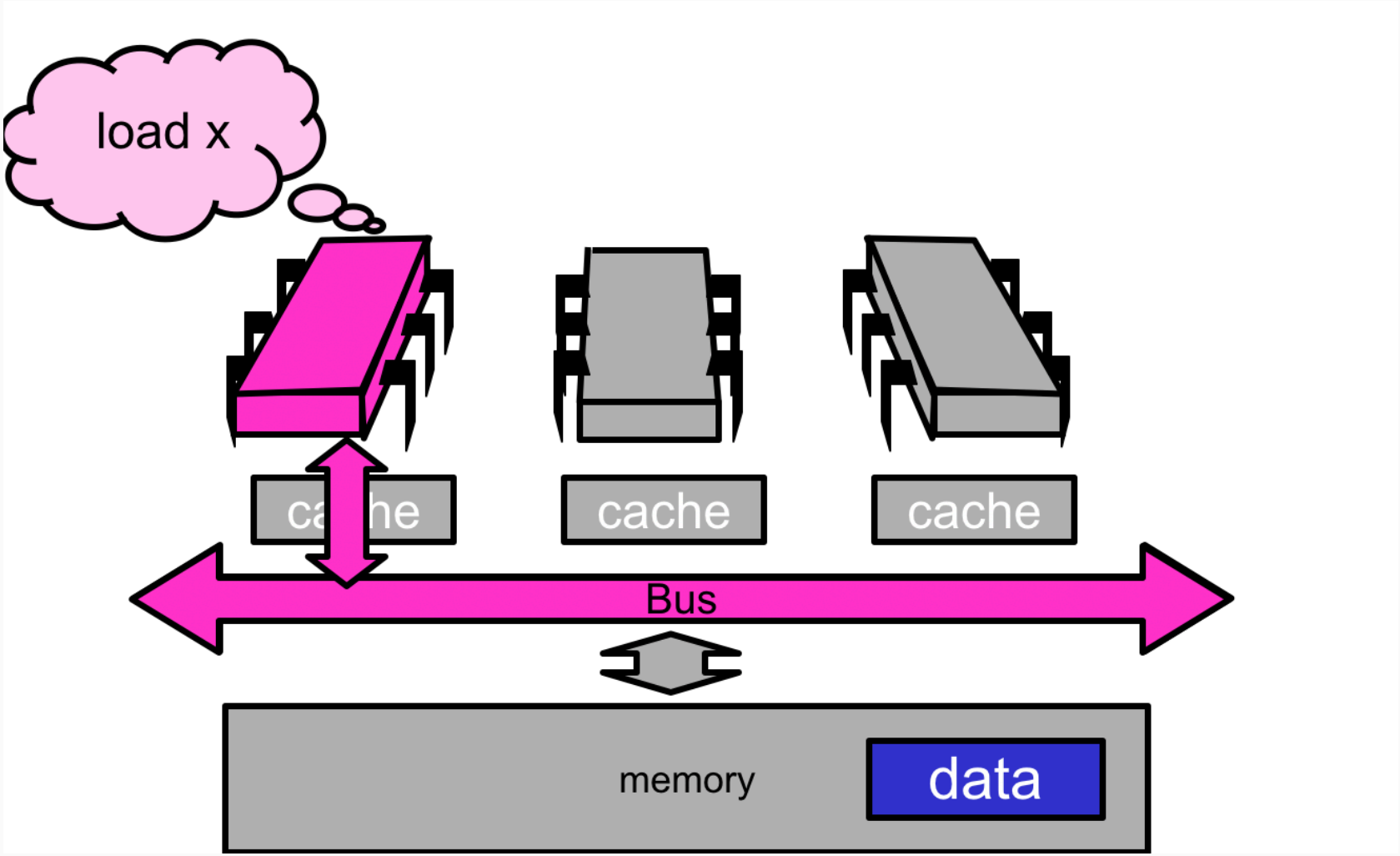
# MESI

MESI (pronouced “messy”) is one of the most commonly used cache coherence protocols. Provides four states that a cache line can be in:

- Modified(M) - Modified, A processor has modified cached data, must write back to memory
- Exclusive(E) - Not modified, Only one processor has a copy of a main memory data in a cache line.
- Shared(S) - Shared Not modified, a piece of data from main memory may be in different caches.
- Invalid(I) - Cache line contents not meaningful

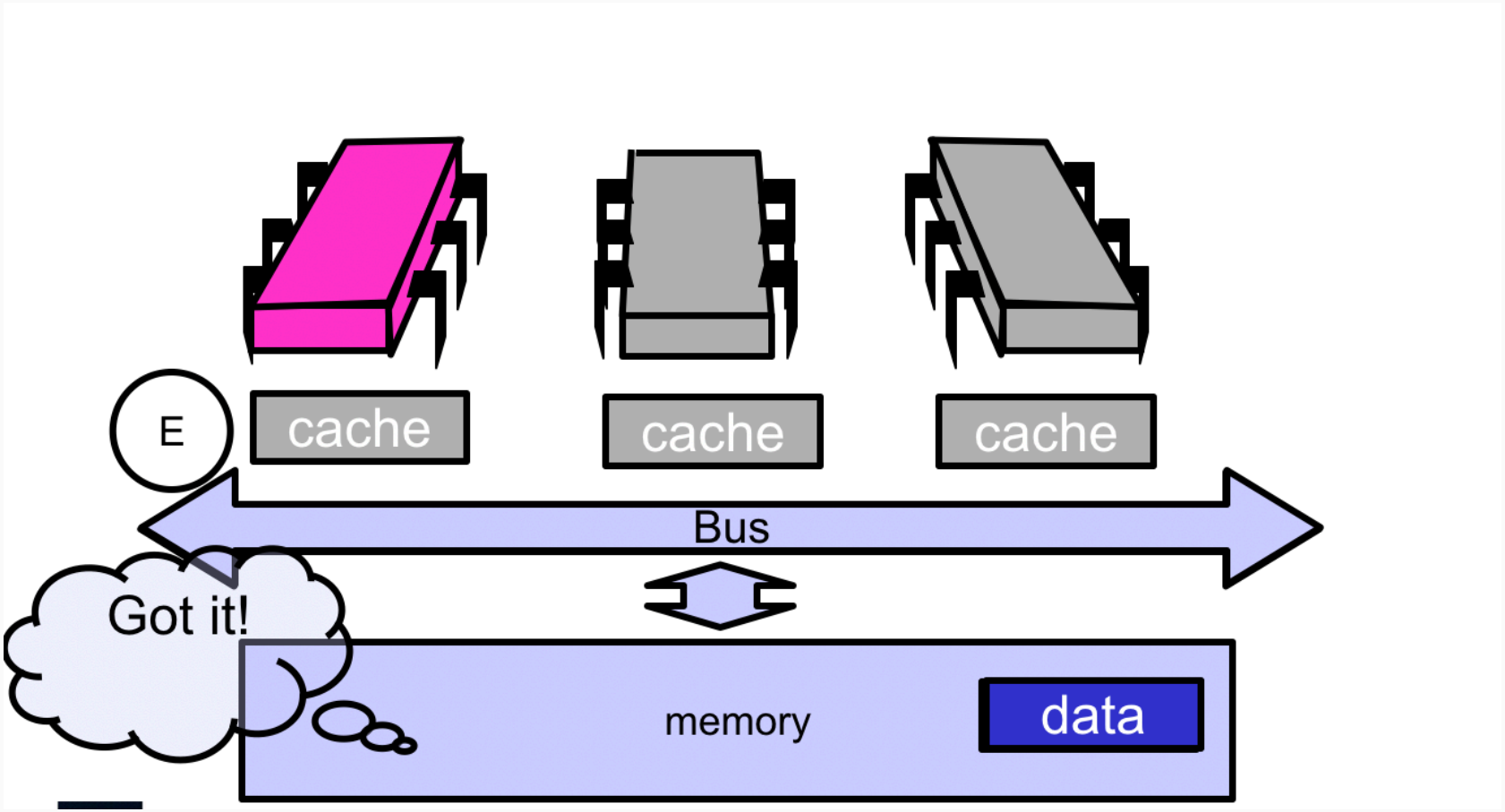
# MESI Example

A processor issues load request at address **x** from main memory.



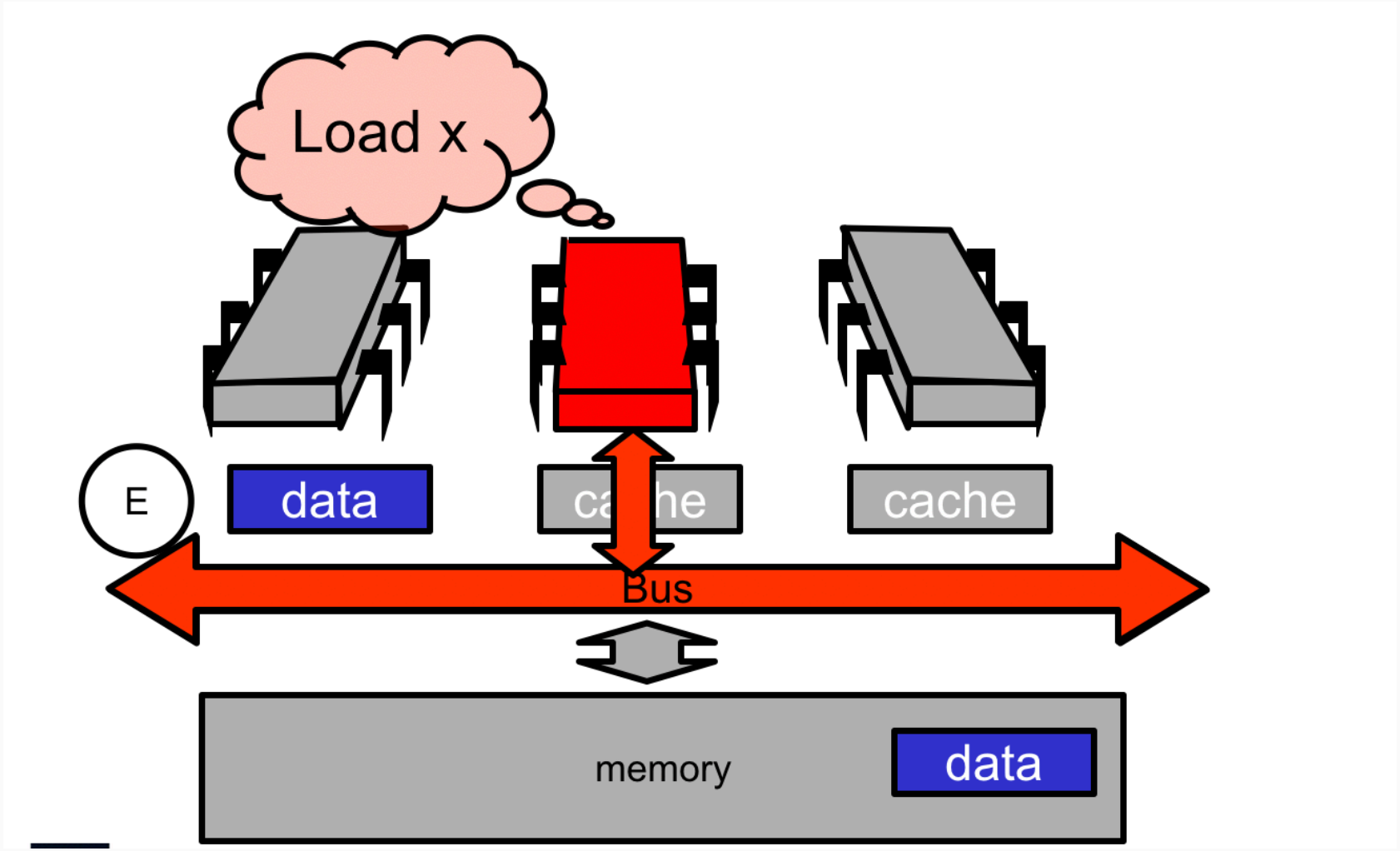
# MESI Example

Main memory responds with the requested data and the processor places the data in a cache line. The data is exclusive to the cache of the requesting processor.



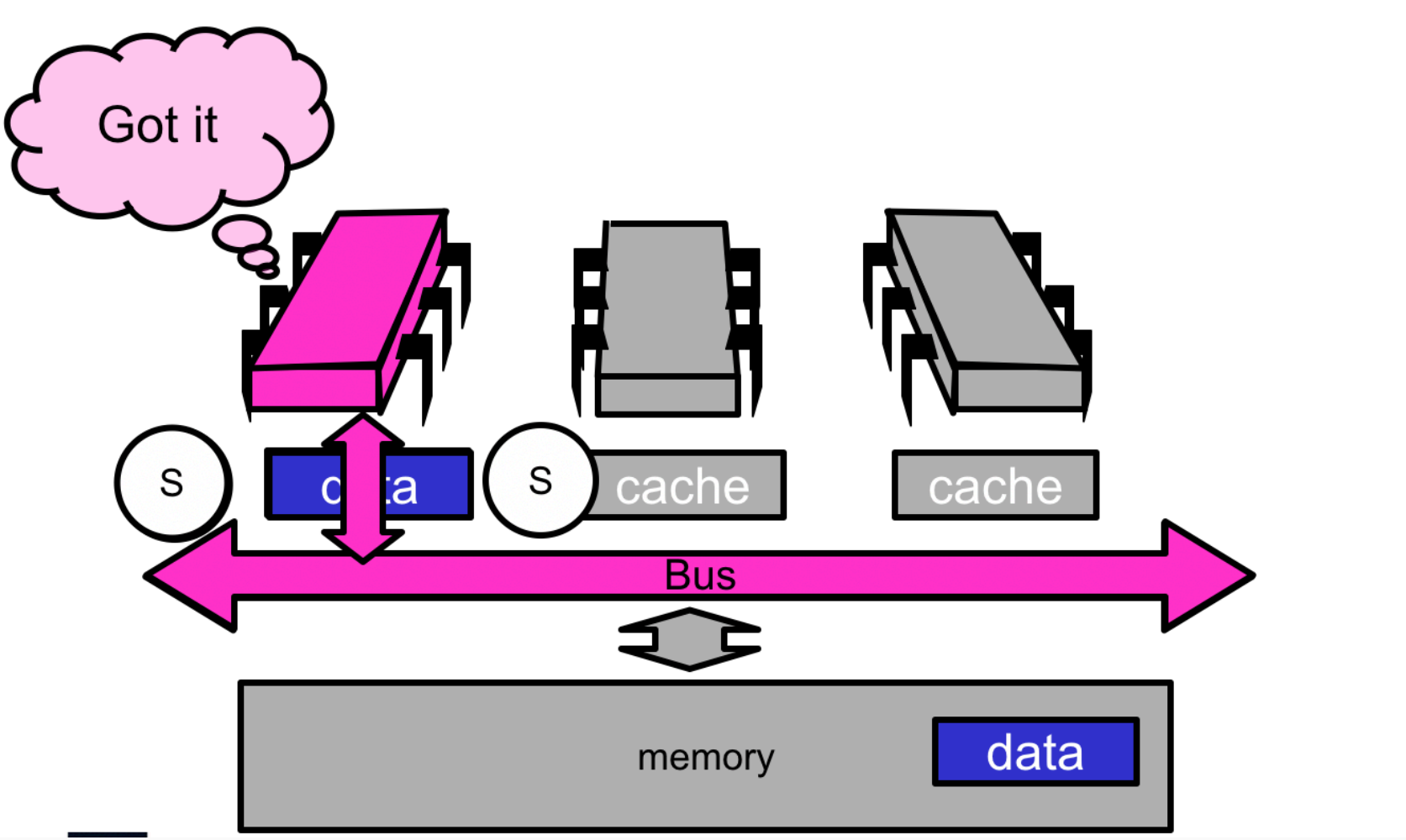
# MESI Example

Another processor requests the data from the same address **x** from main memory.



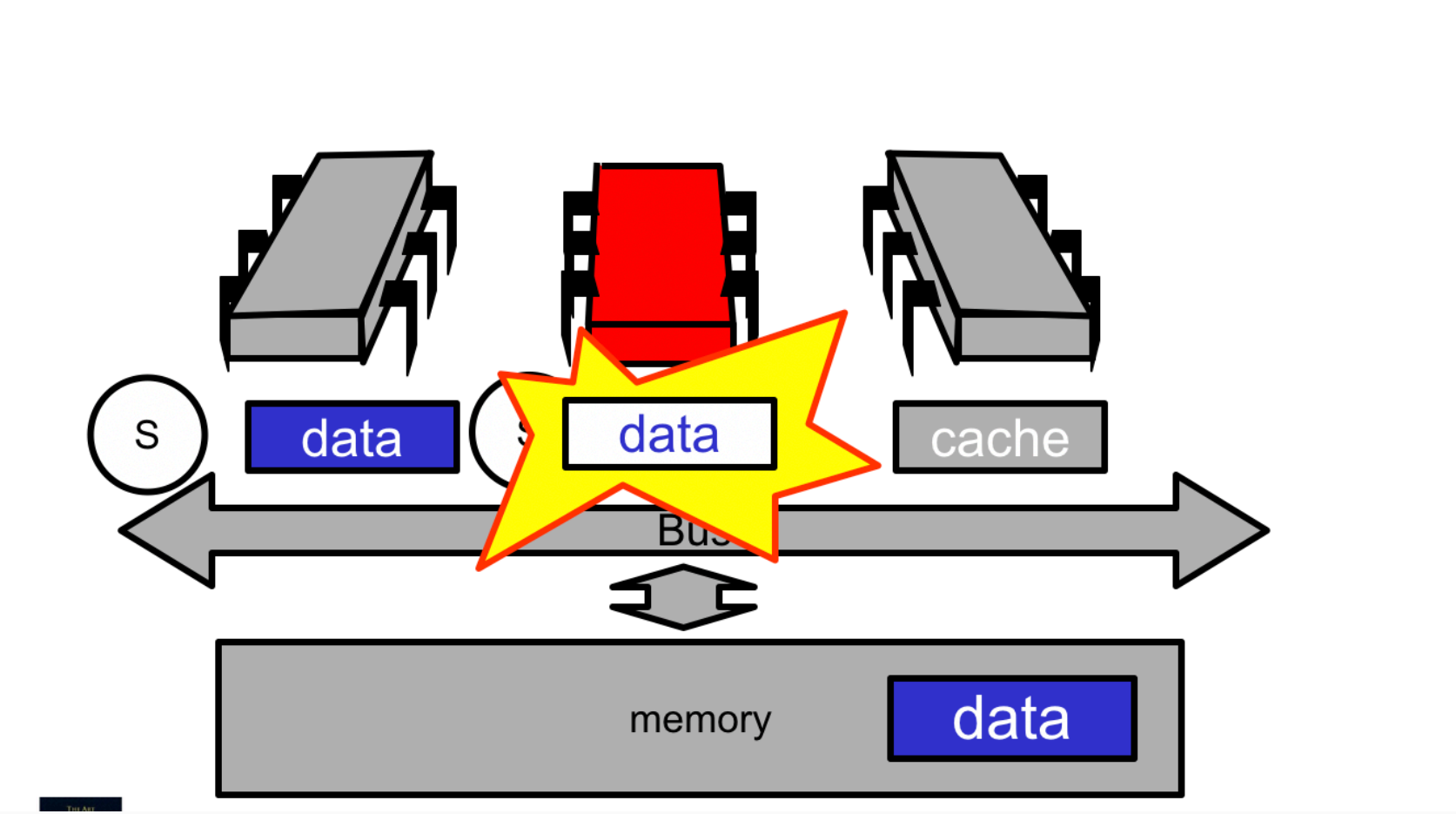
# MESI Example

Processors communicate with each other to update their states to shared since both contain data from the same memory address.



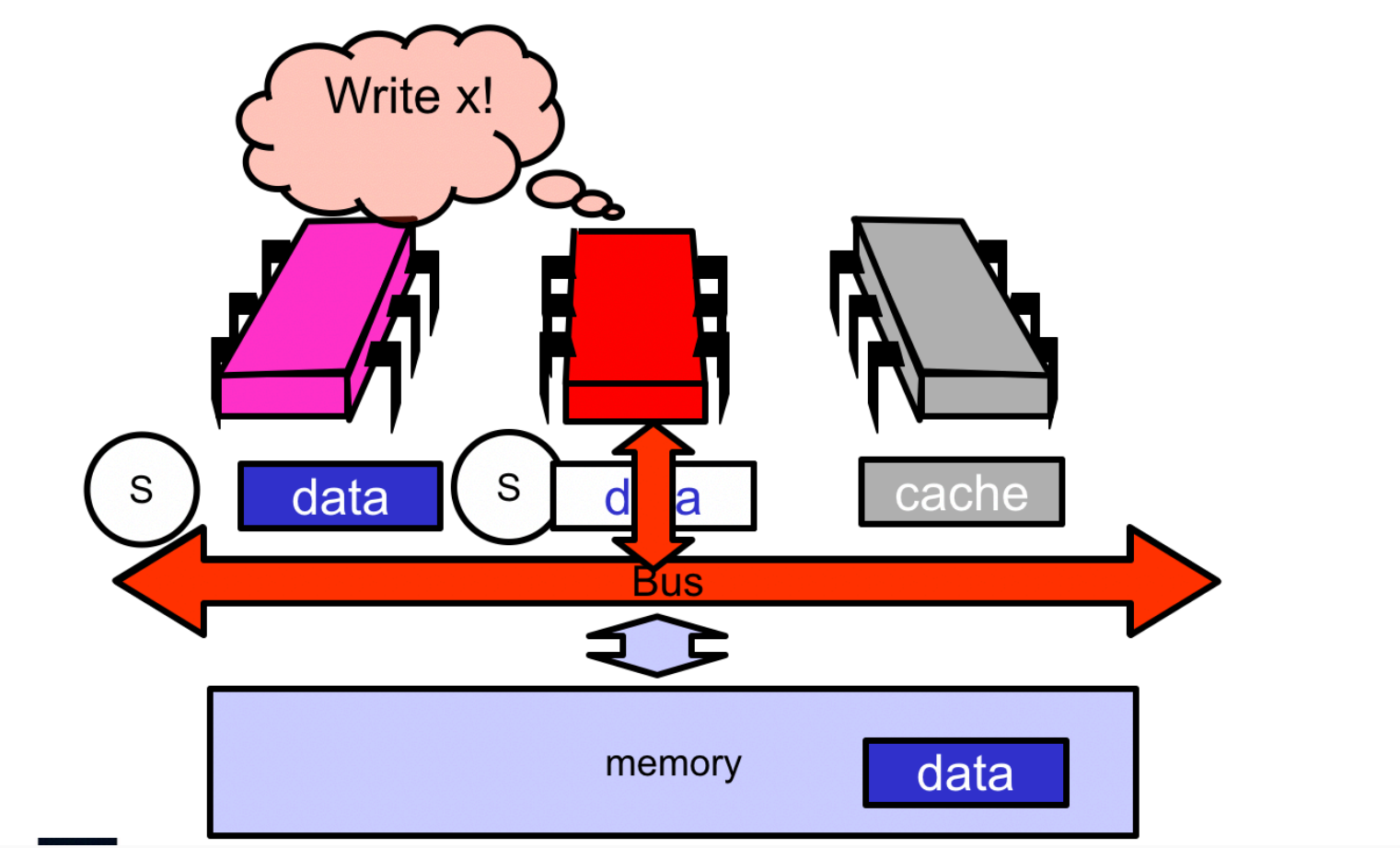
# MESI Example

A processor modifies the data for that memory address in its cache. Other processors need to be notified about this update.



# MESI Example

All other processors are given the updated data and the data is written immediately back to main memory. May not always immediately write data back to main memory. Depends on the write scheme for the cache.



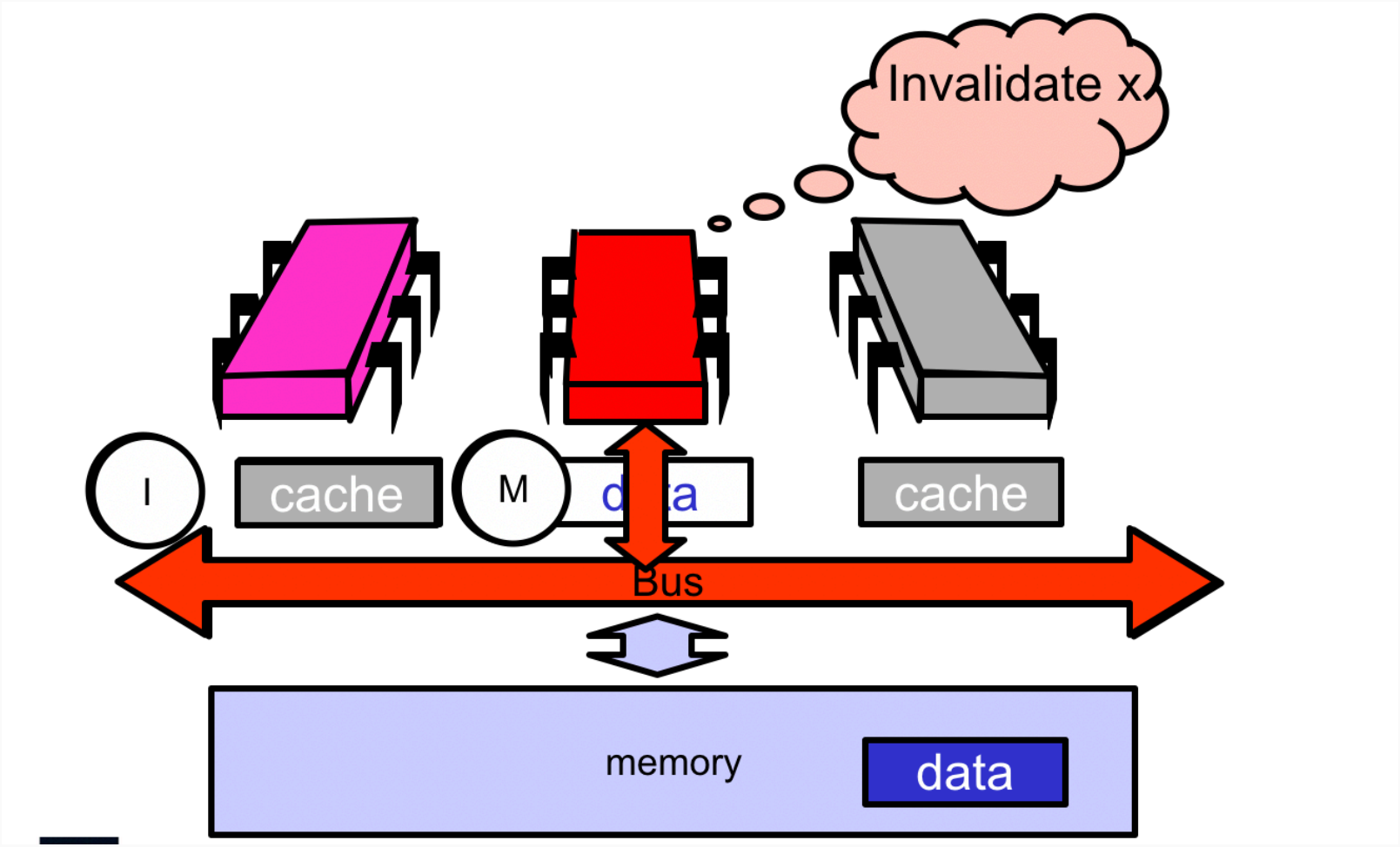
# Write-Through Caches

- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes ...
- Hardly used in practice due to the bus traffic problem

# Write-Back Caches

- Immediately broadcast changes
- Caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.
- The dirty data is usually held in a write-buffer that will eventually write its contents back to main memory periodically.

# MESI - Invalidate



# Programming on Shared Memory Systems

# Shared Memory Model in Go

In Go, each concurrently executing activity/task is called a goroutine:

- A goroutine is *similar* to a thread in other languages/operating systems but it's not actually a thread.
- At program startup, the only goroutine is the one that calls the `main` function, which is known as the main goroutine.

New goroutines are created by the **go** statement:

- Is an ordinary function/method call prefixed by the keyword **go**.
- Causes the function to be called in a new created goroutine
- It returns immediately

```
f() // call f(); wait for it to return
go f() // create a new goroutine that falls f(); don't wait
```

# Goroutines Example: Spinner

```
func main() {
    go spinner(100 * time.Millisecond)
    const n = 45
    fibN := fib(n) // slow
    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}
func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}
func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}
```

# Shared-Memory Programming

As we mentioned earlier, mulitprocessor programming is challenging because modern systems are inherently asynchronous:

- Activities can be halted and delayed without warning by interrupts, preemption, cache misses, failures or other events.
- Interruptions vary enormously in scale:
  - Cache Miss - 10 instructions
  - Page Fault - a few million instructions
  - OS preemptions - hundreds of millions of instructions.
- These situations can lead to race conditions as mentioned earlier.

# Critical Sections

To ensure determinism and avoid race conditions, you need to determine critical sections in your code:

- A critical section is a block of code where potentially more than one thread can execute the code at the same time. This potentially where shared resources are accessed/modified.
- The execution of code in a critical section should, effectively, be executed as serial code.
- Eventually, another thread should be able to access this section once one thread has completed the critical section.

# Low-level Hardware Synchronization Primitives

Many of the low-level synchronization primitives (e.g., locks, monitors, etc.) are built off of specialized hardware primitives/instructions (also known as atomic operations):

- On a shared memory system, an operation is consider atomic if it completes in a single step relative to other threads.
- No other thread can observe the modification to that shared variable half-way through its operation.

# Overview of Synchronization Primitives

Hardware provides simple low-level atomic operations:

- x86 load and store of words
- Special instructions:
  - compare-and-swap (CAS) (AMD, Intel, Sun)
  - pair of test-and-set instructions: loaded-linked and store-conditional (LL/SC) (ARM, IBM PowerPC, etc.)
- Provided in Go: `import "sync/atomic"`

We use those simple low-level atomic operations to build higher-level synchronization primitives:

- Lock (Next week)
- Monitor
- Semaphore
- Conditional Variable
- Barrier

# Problems with Atomic Operations

Best practices is to use atomic operations sparingly because:
Problem with atomic operations:

- Most atomic operations are implemented using CAS or (LL/SC), which take significant more cycles to complete than a simple load or store instruction.
- Causes a memory fence, which forces the write-back buffer to be sent to main memory. This process can then stall other processors from reading/writing to main memory
- Prevents out-of-order execution and various compiler optimizations.
- Cost to performance varies depending on architectures, program design, etc.
- Adds more hardware complexity.