SRM

INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR, Chennai.

18CSC203J - COA

PARALLELISM

Course Learning Rationale (CLR)

CLR-4 : *Study about parallel processing and performance considerations.*

Course Learning Outcome (CLO)

CLO-4 : *Analyze concepts of parallelism and multi-core processors*

Contents

- Parallelism
- Need for Parallelism
- Types of Parallelism
- Applications of Parallelism
- Parallelism in Software
  - Instruction Level Parallelism
  - Data Level Parallelism
- Challenges in Parallelism
- Architecture of Parallel System
  - Flynn's Classification
  - SISD , SIMD
  - MISD, MIMD
- Hardware Multi Threading
  - Coarse Grain Parallelism
  - Fine Grain Parallelism
- Uni-Processor and Multi Processor
- Muti-Core Processor
- Memory in Multi-Processor System
- Cache Coherency in Multi-Processor System
- MESI Protocol for Multi-Processor System

Parallelism

- Executing two or more operations at the same time is known as parallelism.

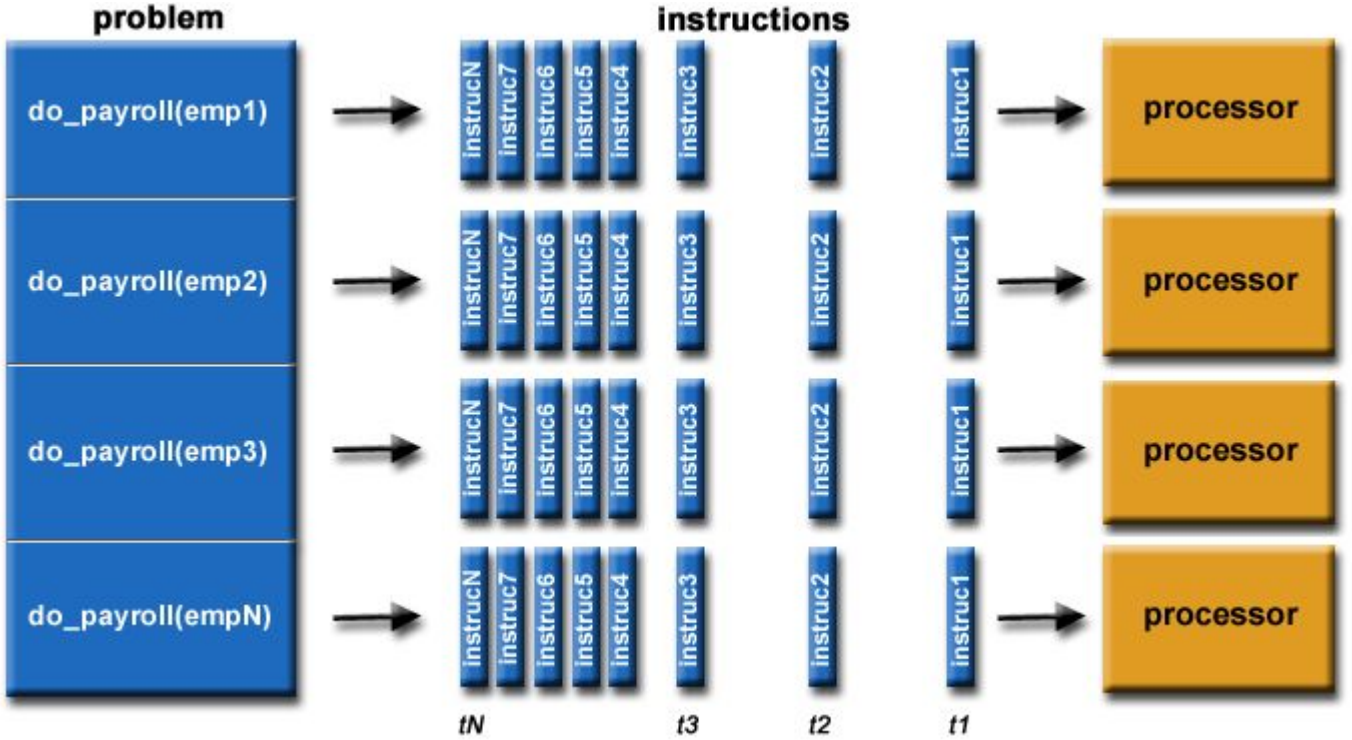


Image Courtesy : https://computing.llnl.gov/tutorials/parallel_comp/

Parallel Processing

- Parallel processing improves the performance by executing two or more instructions simultaneously
- A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem.
- On a single processor may have Two or more ALUs which work concurrently to increase throughput
- The system may have two or more processors operating concurrently (Eg: i3,i5,i7)

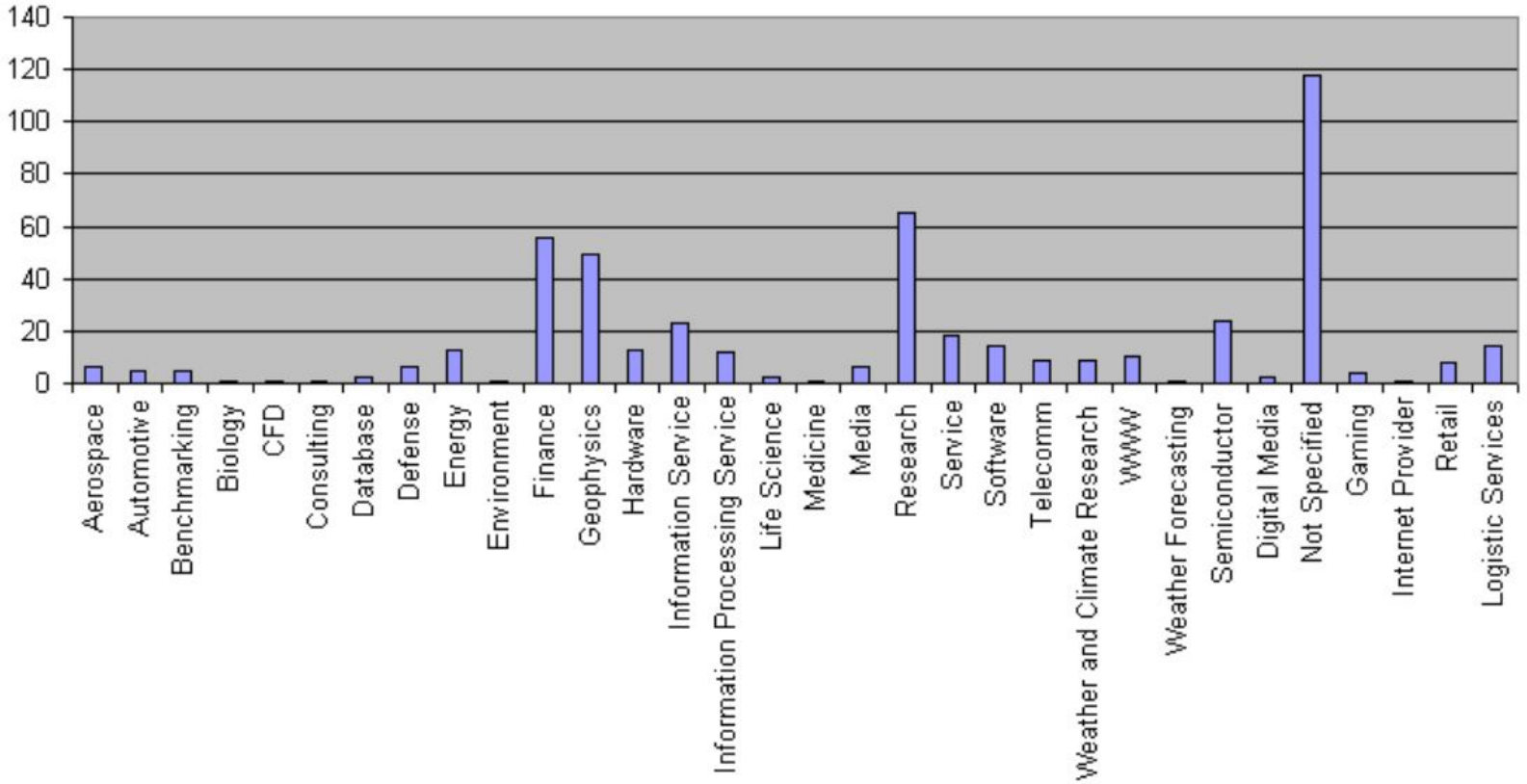
Goals of Parallelism

- To increase the computational speed (ie) to reduce the amount of time that you need to wait for a problem to be solved
- To increase throughput (ie) the number of processes completed on unit time
- To improve the performance of the computer for a given clock speed
- To solve bigger problems that might not fit in the limited memory of a single CPU
- To take advantage of non-local resources when the local resources are finite

Applications of Parallelism

- Numeric Weather Prediction
- Socio Economics
- Finite Element Analysis
- Artificial Intelligence and Automation
- Genetic Engineering
- Weapon Research and Defense
- Medical Applications
- Remote Sensing Applications

Applications of Parallelism



Types of Parallelism

1. Hardware Parallelism
2. Software Parallelism

- Hardware Parallelism :

It is based on the processor multiplicity and machine architecture.

One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle. If a processor sends k instructions per machine cycle, then it is called a k-issue processor

Software Parallelism

- Determined by the dependency among the sub tasks
- The flow control graph used to represent the degree of parallelism in the program
- The flow graphs shows the possible ways in which the sub programs can be executed in parallel.
- Software parallelism can be exhibited by the algorithm, programming pattern (threads) and optimization on compilers
- Parallelism in a program varies during the execution period .
- It limits the sustained performance of the processor.

Example

Example:

A=L1 * L2 + L3 * L4

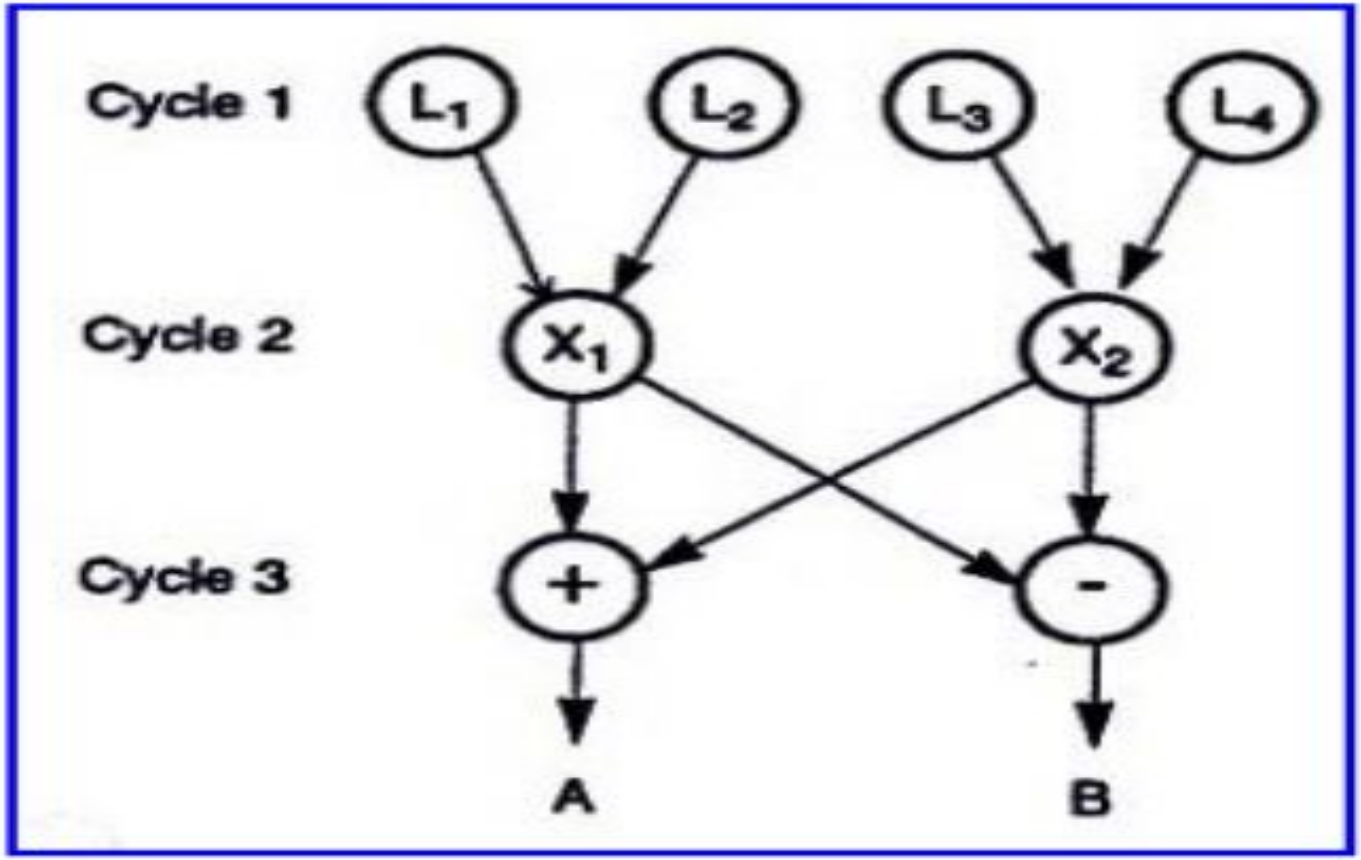
B=L1 * L2 - L3 * L4

Software Parallelism:

There are 8 instructions

- FOUR Load instructions(L1,L2,L3&L4).
- TWO Multiply instructions(X1&X2)
- ONE Add instruction(+)
- ONE Subtract instruction(-)

The parallelism varies from 4 to 2 in three cycles.



$$\text{Average S/W Parallelism} = \frac{8 \text{ cycles}}{3 \text{ cycles}} = \frac{8}{3} = 2.67$$

Hardware Parallelism - Example

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
—Deemed to be University—

Parallel Execution:

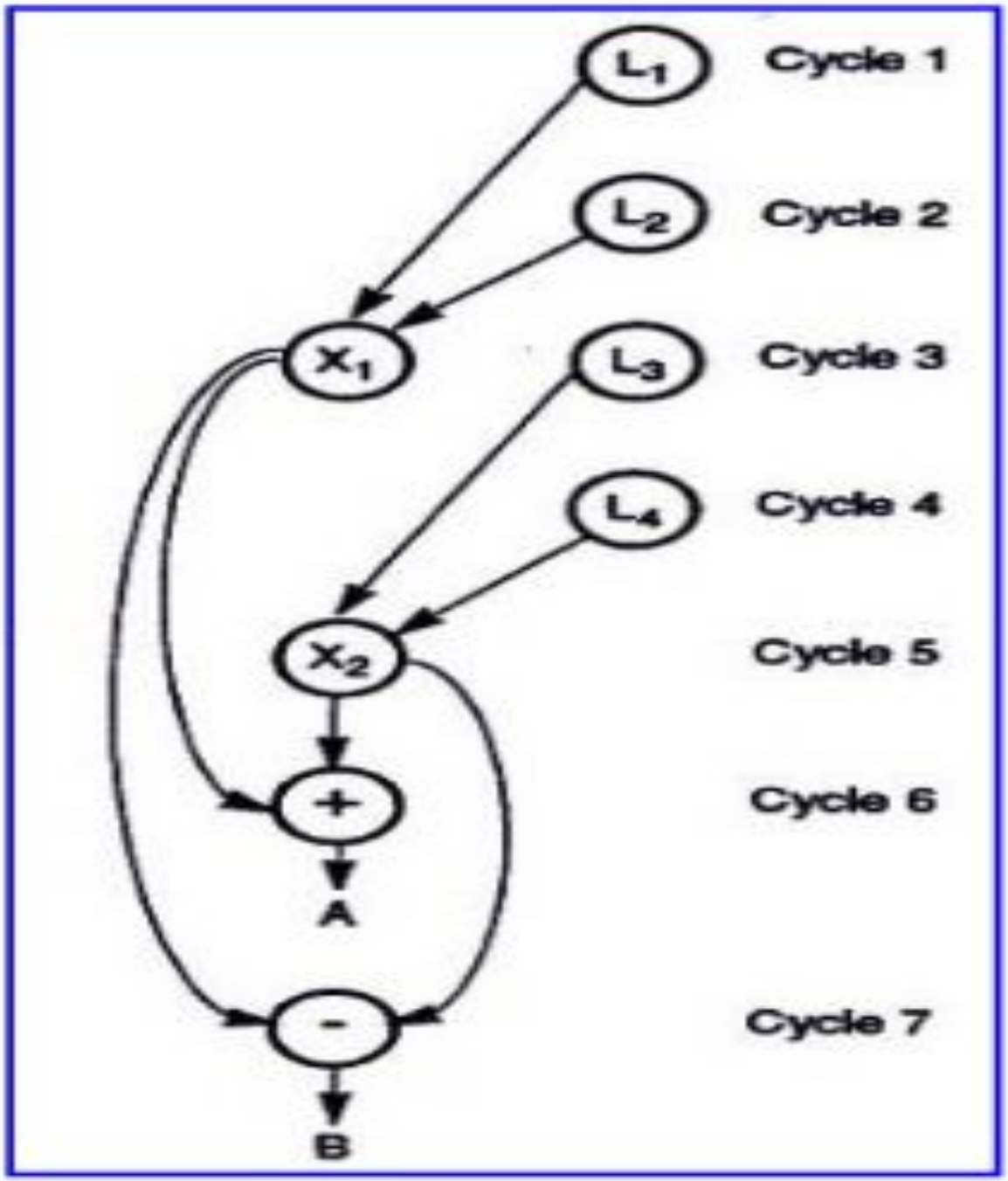
Using TWO-issue processor:

The processor can execute one memory access(Load / Store) and one arithmetic operation(multiply/add/subtract) simultaneously.

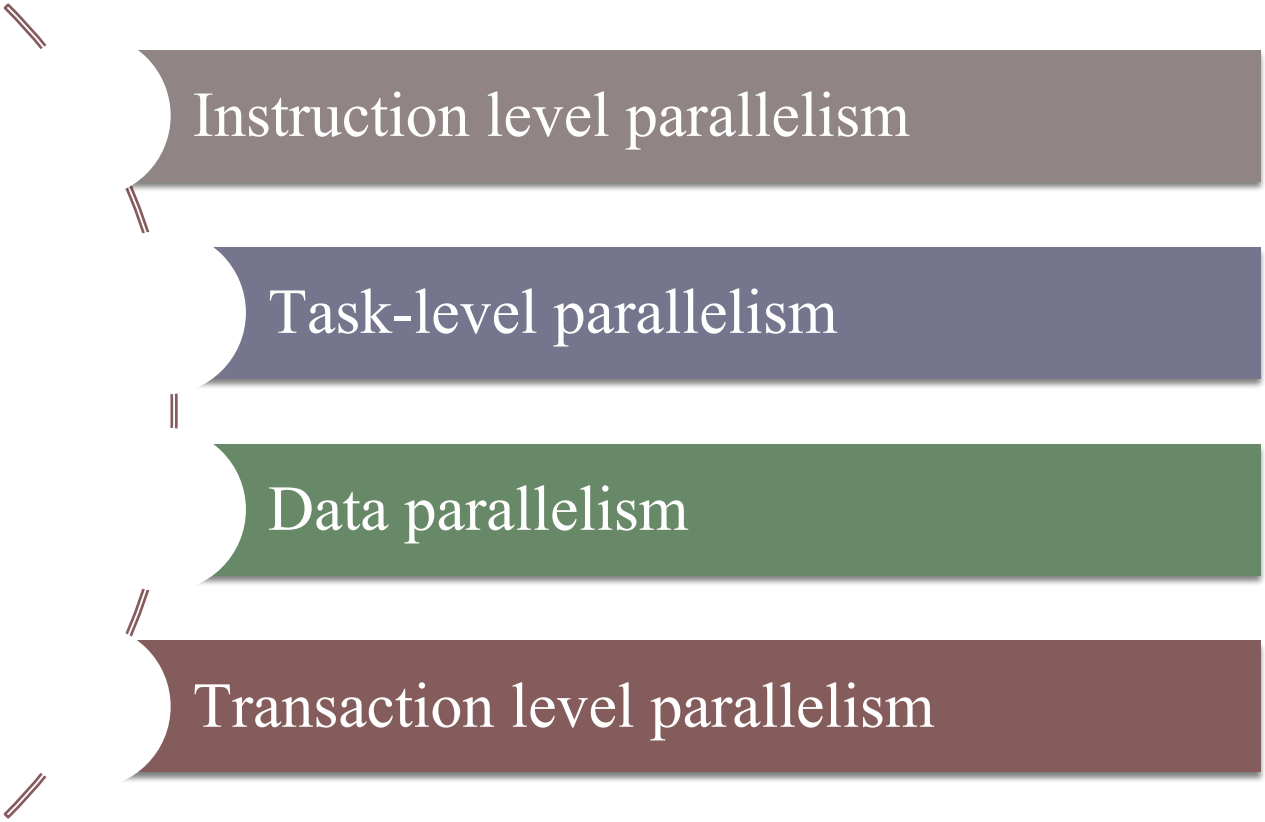
The program must execute in 7 cycles.

The H/w parallelism average is 8/7=1.14.

It is clear from this example that there is a mismatch between the s/w and h/w parallelism.


L1
Cycle 1
L2
Cycle 2
X1
L3
Cycle 3
L4
Cycle 4
X2
Cycle 5
+
Cycle 6
A
-
Cycle 7
B

Software Parallelism - Types



Instruction Level Parallelism

- Instruction level Parallelism (ILP) is a measure of how many operations (instructions) can be performed in parallel at the same time in a computer program.
- Parallel instructions are set of instructions that do not depend on each other for execution
- ILP allows the compiler and processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

Eg. Instruction Level Parallelism

Consider the following example

1. x= a+b
2. y=c-d
3. z=x * y

Operation depends on the results of 1 & 2

So ‘Z ‘ cannot be calculated until X & Y are calculated

But 1 & 2 do not depend on any other. So they can be computed simultaneously.

- If we assume that each operation can be completed in one unit of time then theses 3 operations can be completed in 2 units of time .
- ILP factor is 3/2=1.5 which is greater than without ILP.
- A superscalar CPU architecture implements ILP inside a single processor which allows faster CPU throughput at the same clock rate.

Instruction-level parallelism (ILP)

Executes the multiple instructions on same instruction streams simultaneously.

These ae generated and managed by either hardware (superscalar) or by compiler (VLIW) – Limited in practice by data and control dependences

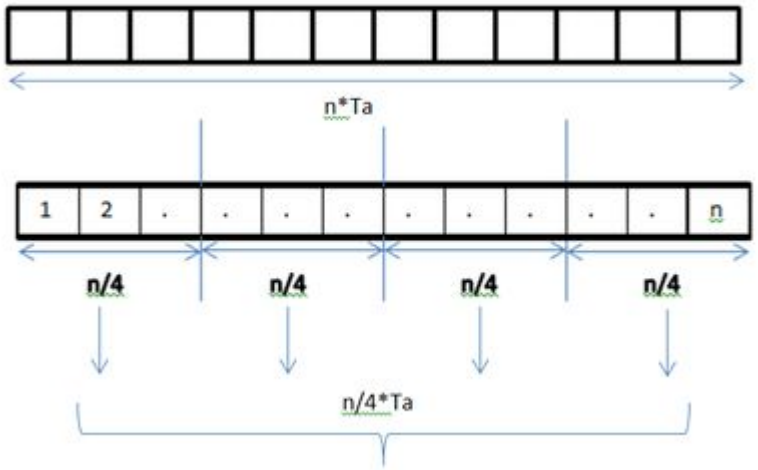
Data-Level Parallelism (DLP)

- **Data parallelism** is parallelization across multiple processors in **parallel computing** environments.
- It focuses on distributing the **data** across different nodes, which operate on the **data** in **parallel**.
- Instructions from a single stream operate concurrently on several data
- Limited by non-regular data manipulation patterns and by memory bandwidth

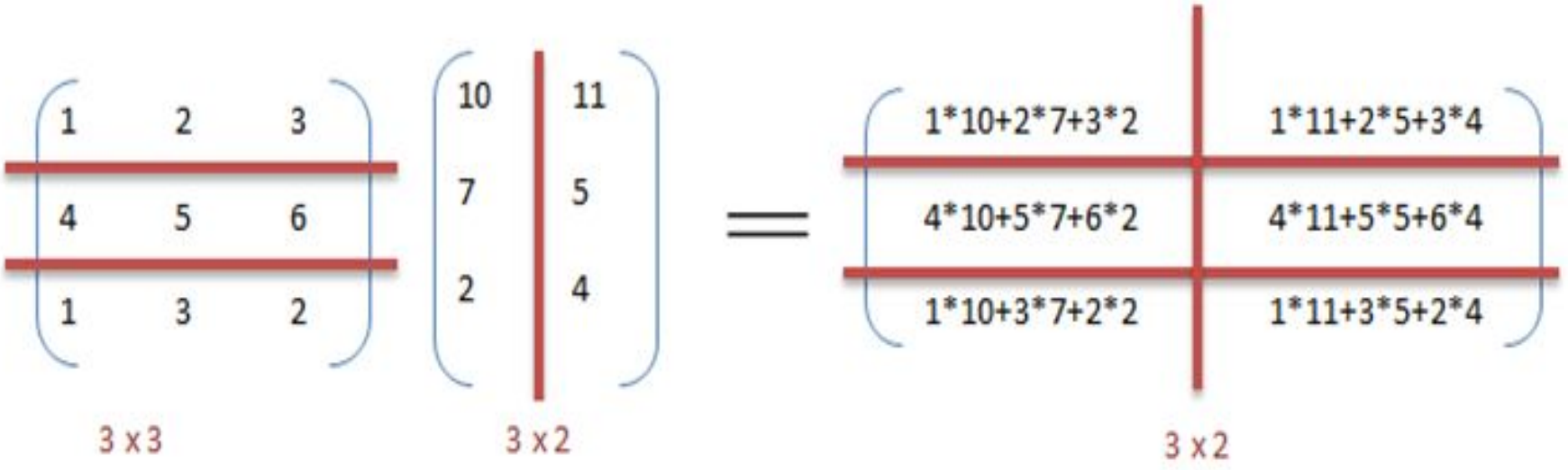
DLP - example

- Let us assume we want to sum all the elements of the given array of size n and the time for a single addition operation is Ta time units.
- In the case of sequential execution, the time taken by the process will be n*Ta time unit
- if we execute this job as a data parallel job on 4 processors the time taken would reduce to (n/4)*Ta + merging overhead time units.

DLP in Adding elements of array



DLP in matrix multiplication



- O(m*n*k) when executed in parallel using *m***k* processors.

- The locality of data references plays an important part in evaluating the performance of a data parallel programming model.
- Locality of data depends on the memory accesses performed by the program as well as the size of the cache.

Thread-Level or Task-Level Parallelism (TLP)

- Multiple program/Task threads are created for same application and executed simultaneously

- These task threads are created and managed by compiler and hardware, however the program threads are created by programmer and managed by compiler and hardware during run time.

- User rarely since suffers by communication/ synchronization overheads

Challenges under Parallel Processing

- The Hardware Model
- Limitations on the Window Size and Maximum Issue Count
- The Effects of Realistic Branch and Jump Prediction
- The Effects of Finite Registers
- The Effects of Imperfect Alias Analysis

Courtesy ; https://facultytalkies.com/courses/cs8491-computer-architecture-online-video-course/lessons/parallel-processing/

Flynn's Classification

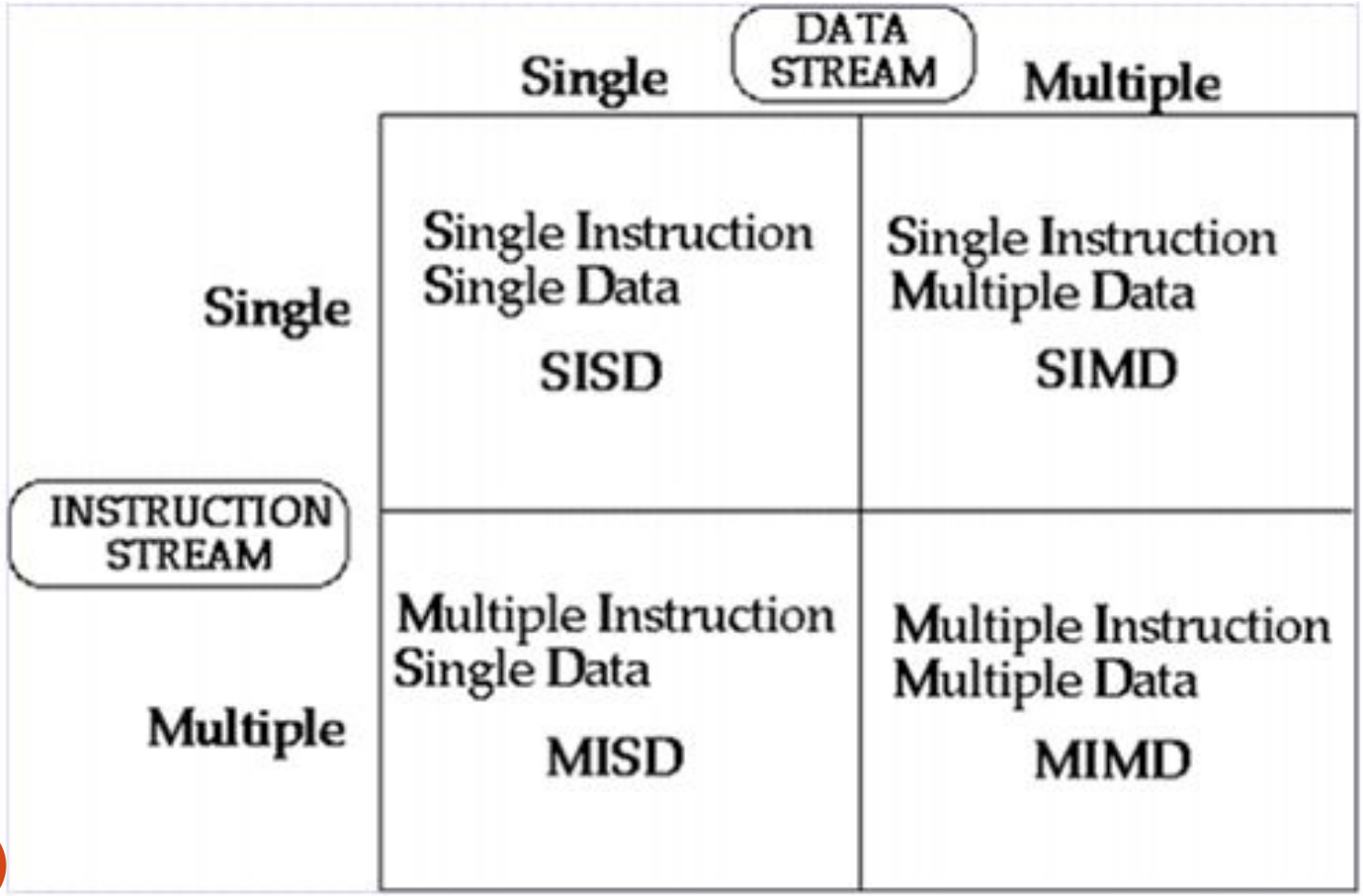
- Was proposed by researcher Michael J. Flynn in 1966.
- It is the most commonly accepted taxonomy of computer organization.
- In this classification, computers are classified by whether it processes a single instruction at a time or multiple instructions simultaneously, and whether it operates on one or multiple data sets.

Flynn's Classification

- This taxonomy distinguishes multi-processor computer architectures according two independent dimensions
  - Instruction stream
  - Data stream.
- An instruction stream is sequence of instructions executed by machine.
- A data stream is a sequence of data including input, partial or temporary results used by instruction stream.
- Each of these dimensions can have only one of two possible states: Single or Multiple.
- Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections.

Flynn's Classification

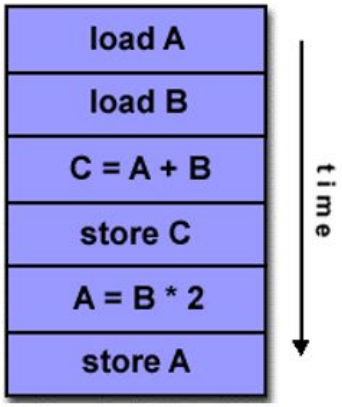
- Four category of Flynn classification

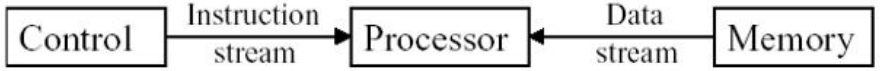


SISD

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Single data: only one data stream is being used as input during any one clock cycle.
- Deterministic execution.
- Instructions are executed sequentially.

- SISD computer having one control unit, one processor unit and single memory unit.
-

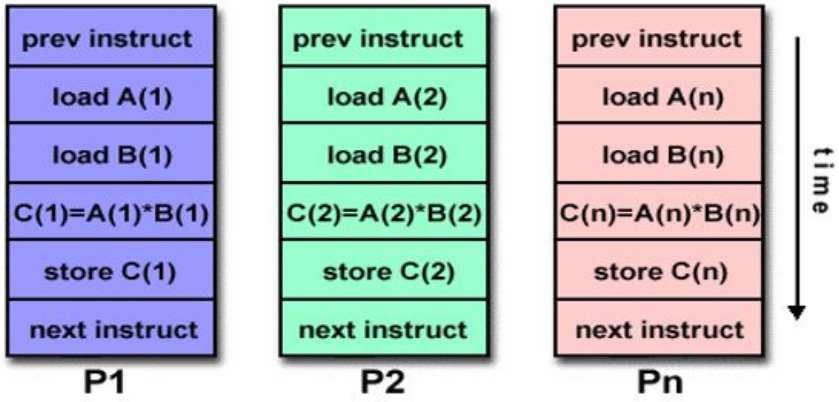


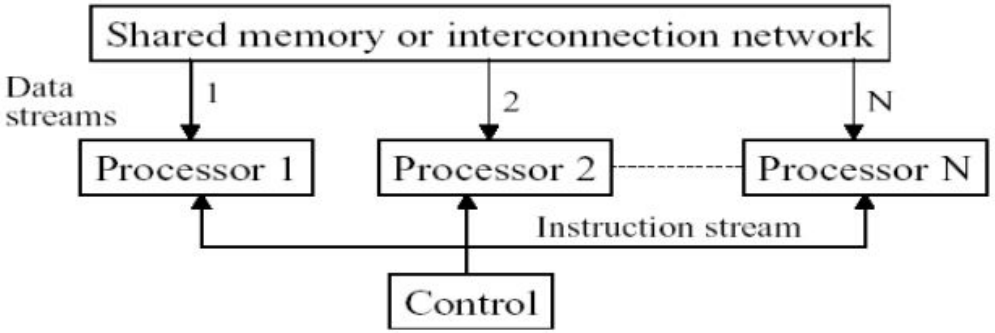


SIMD

- A type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle .
- Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit

- single instruction is executed by different processing unit on different set of data

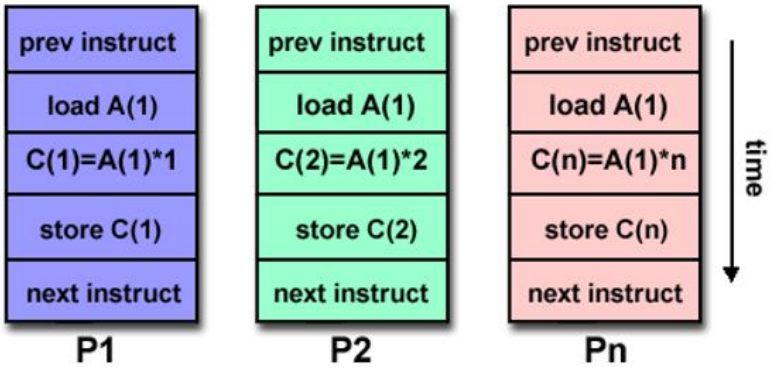


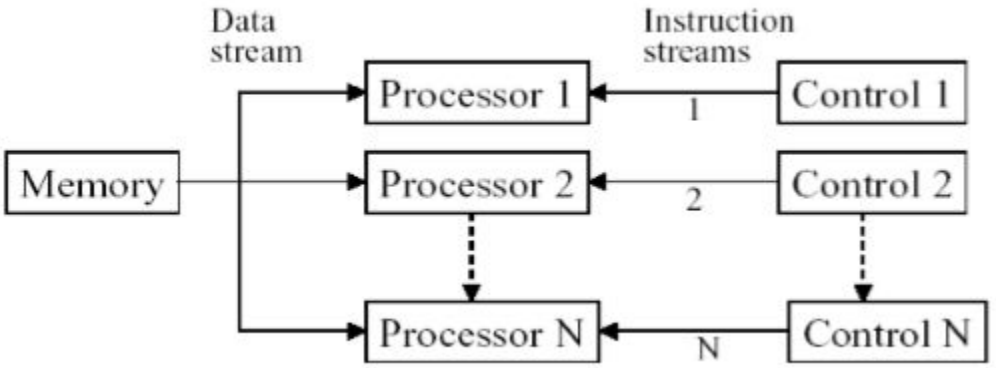


MISD

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction.
- A single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

- same data flow through a linear array of processors executing different instruction streams

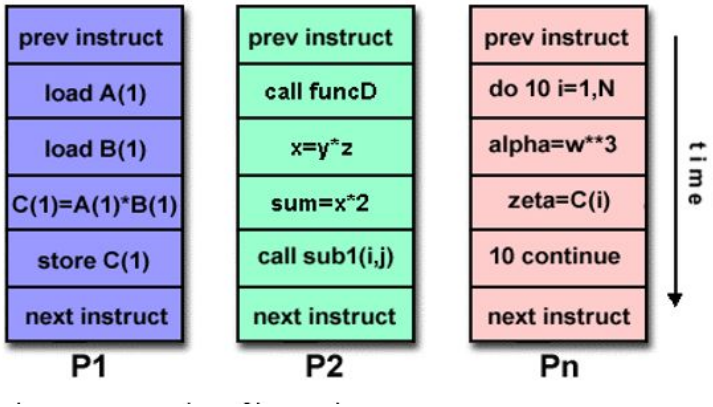


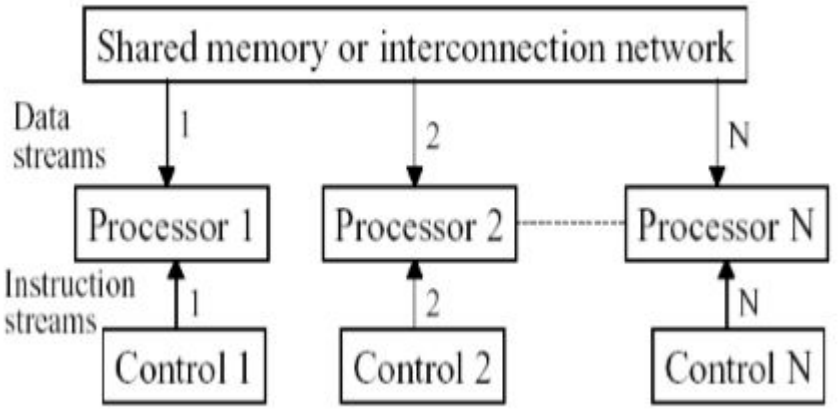


MIMD

- Multiple Instruction: every processor may be executing a different instruction stream.
- Multiple Data: every processor may be working with a different data stream.
- Execution can be synchronous or asynchronous, deterministic or nondeterministic

- Different processor each processing different task.





Thread and Multithreads

- It is an instruction stream with state (Register/Memory)
- The Thread state is called as thread context (Register state Register context)
- It may belongs to same process or different processes
- Threads in the same program share the same address space
- Under Multi threading/ Multitasking when new thread need execution, processor saves the older thread context and takes up the new thread context.

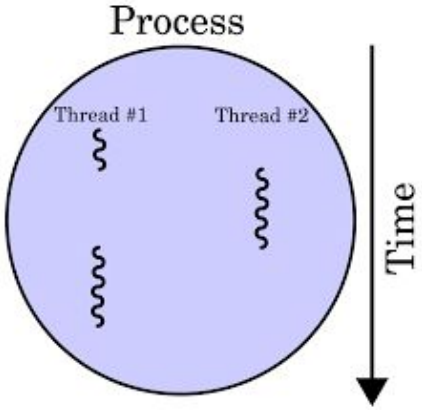
Hardware Multithreading

- Hardware multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion to try to utilize the hardware resources efficiently.

- To permit this sharing, the processor must duplicate the independent state of each thread. It Increases the utilization of a processor

- For example, each thread would have a separate copy of register file and program counter. The memory itself can be shared through virtual memory mechanisms, which already support multi-programming.

- In addition, hardware must supportability to change to a different thread relatively quickly. In particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.



Courtesy: Wikipedia

Multithreading Types

- Fine Grained
- Course Grained
- **Simultaneous**

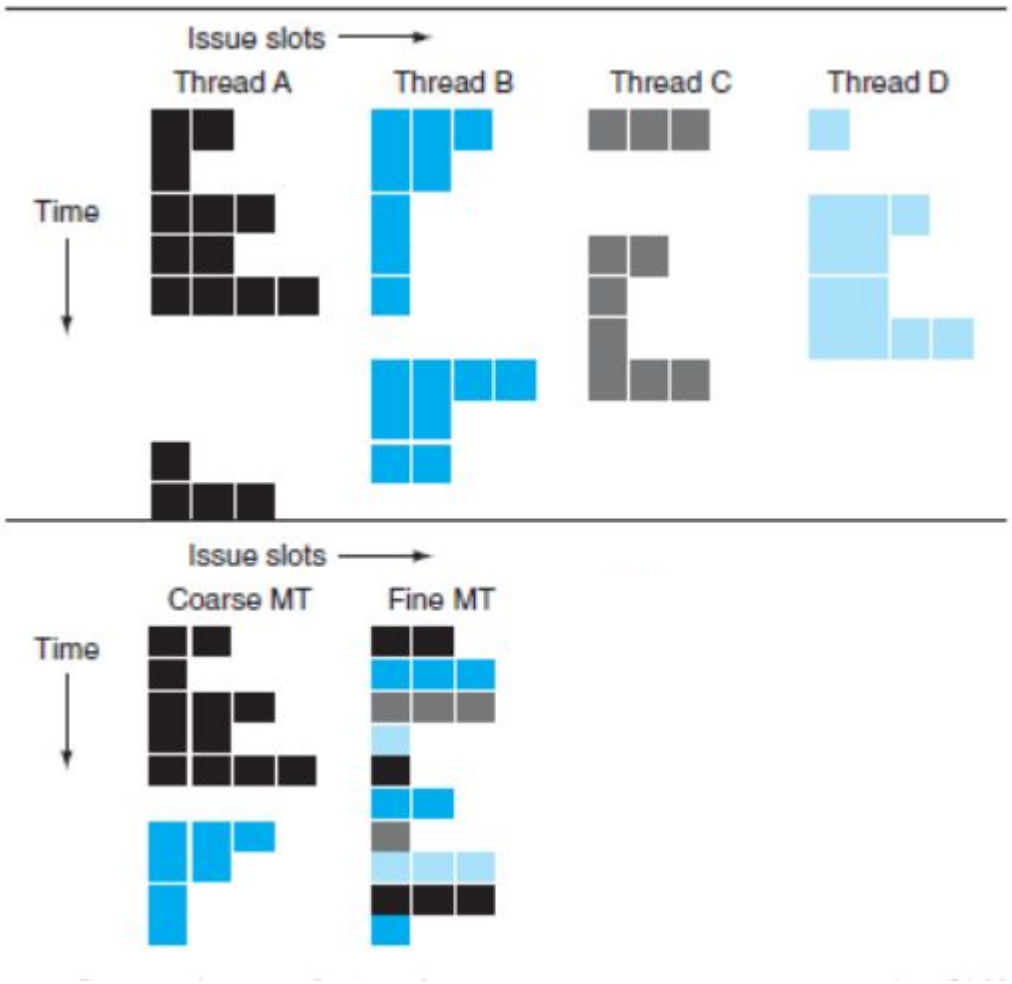
Fine Grained Multi Threading

- Fine-grained multithreading switches between threads on each instruction, resulting in interleaved execution of multiple threads.
-
- This interleaving is often done in around-robin fashion, skipping any threads that are stalled at that clock cycle.
- To make fine-grained multithreading practical, processor must be able to switch threads on every clock cycle.
- One advantage of fine-grained multithreading is that it can hide throughput losses that arise from both short and long stalls, since instructions from or threads can be executed when one thread stalls.
- The primary disadvantage of fine-grained multithreading is that it slows down execution of individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from or threads.

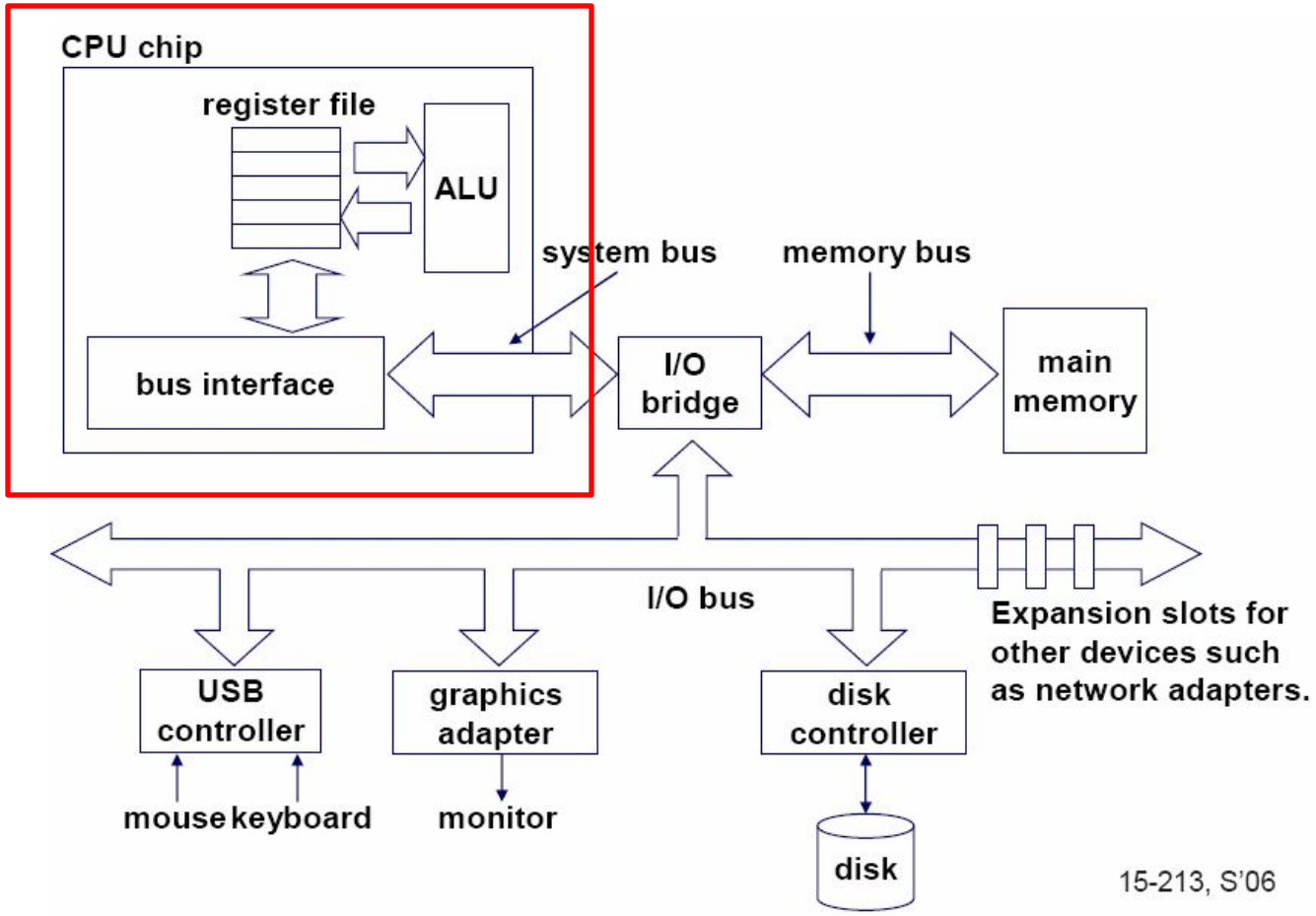
Coarse Grained Multi Threading

- Coarse-grained multithreading was invented as an alternative to fine-grained multithreading.
- Coarse-grained multithreading switches threads only on costly stalls, such as last-level cache misses.
- This change relieves need to have thread switching be extremely fast and is much less likely to slow down execution of an individual thread, since instructions from or threads will only be issued when a thread encounters a costly stall.
- Drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls.
- This limitation arises from pipeline start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, pipeline must be emptied or frozen.
- The new thread that begins executing after stall must fill pipeline before instructions will be able to complete. Due to start-up overhead, coarse-grained multithreading is much more useful for reducing penalty of high-cost stalls, where pipeline refill is negligible compared to stall time.

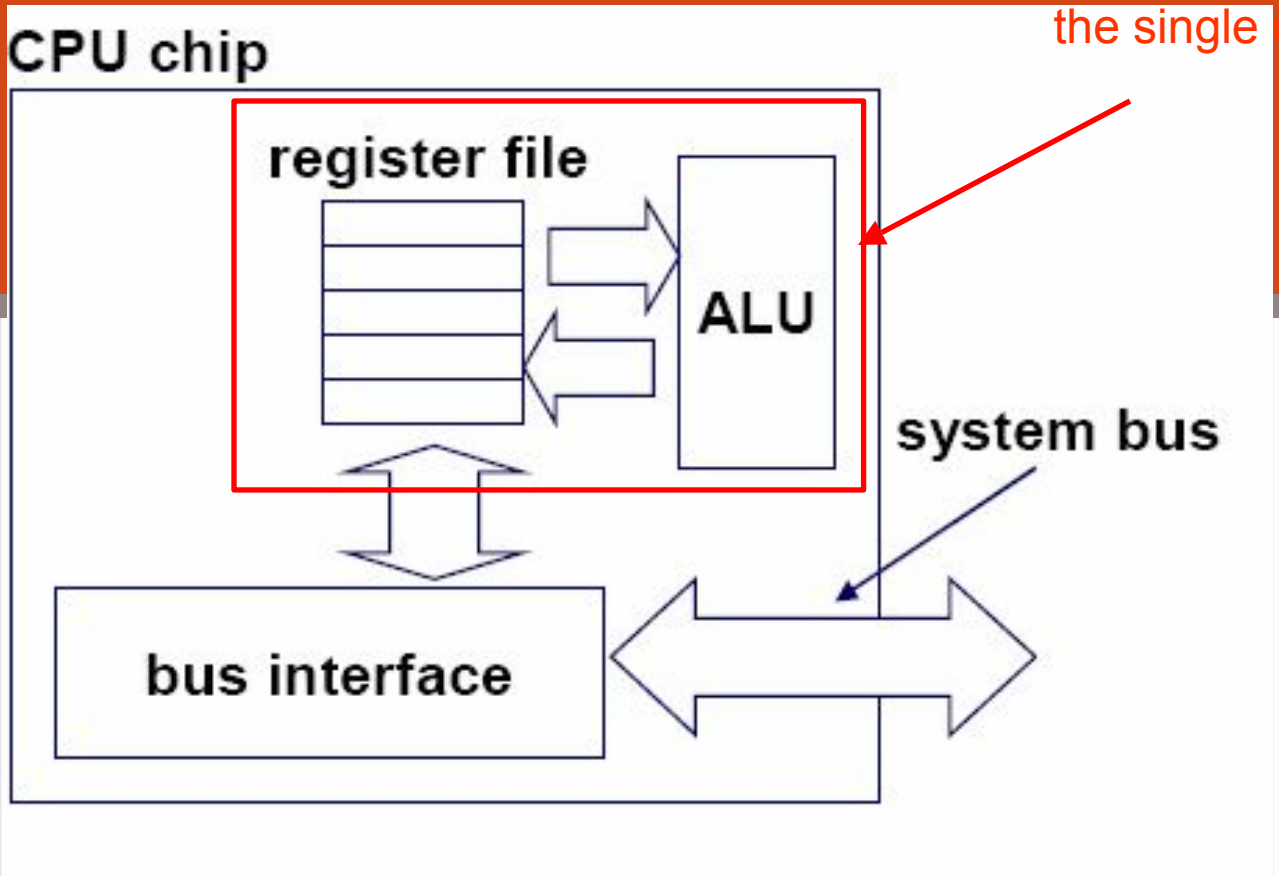
Comparison


Issue slots
Thread A
Thread B
Thread C
Thread D
Time
Issue slots
Coarse MT
Fine MT
Time

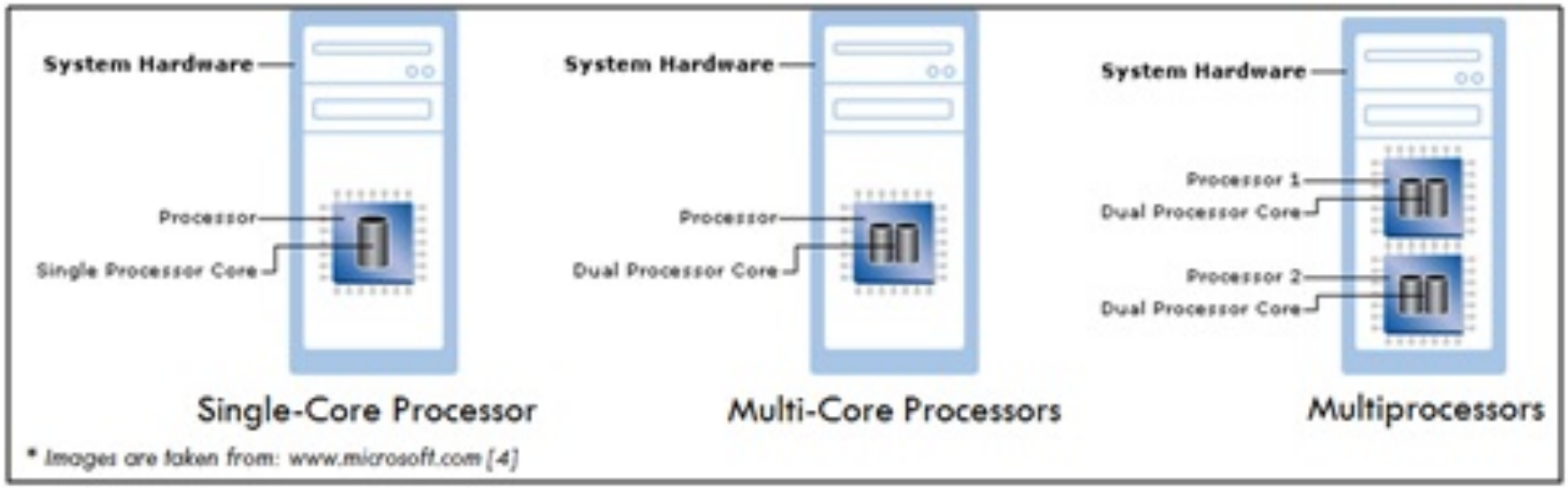
Single-core computer



Single-core CPU chip



Single Vs Multicore Processors

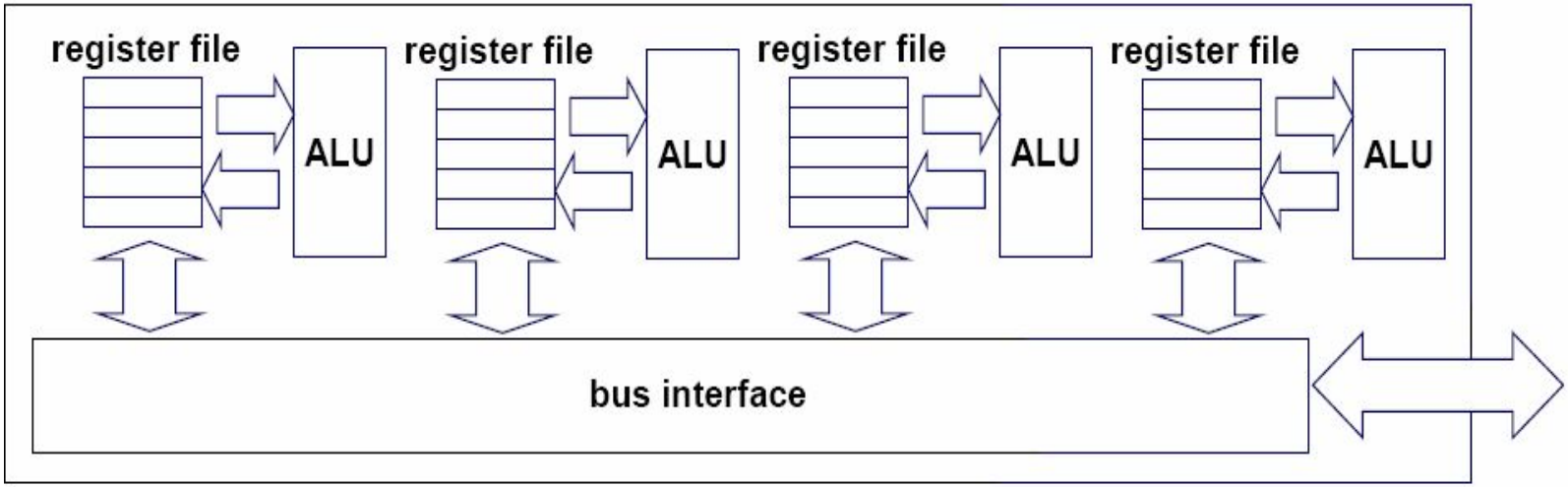


Courtesy:
https://facultytalkies.com/courses/cs8491-computer-architecture-online-video-course/lessons/multi-core-processors/

Multi-core architectures

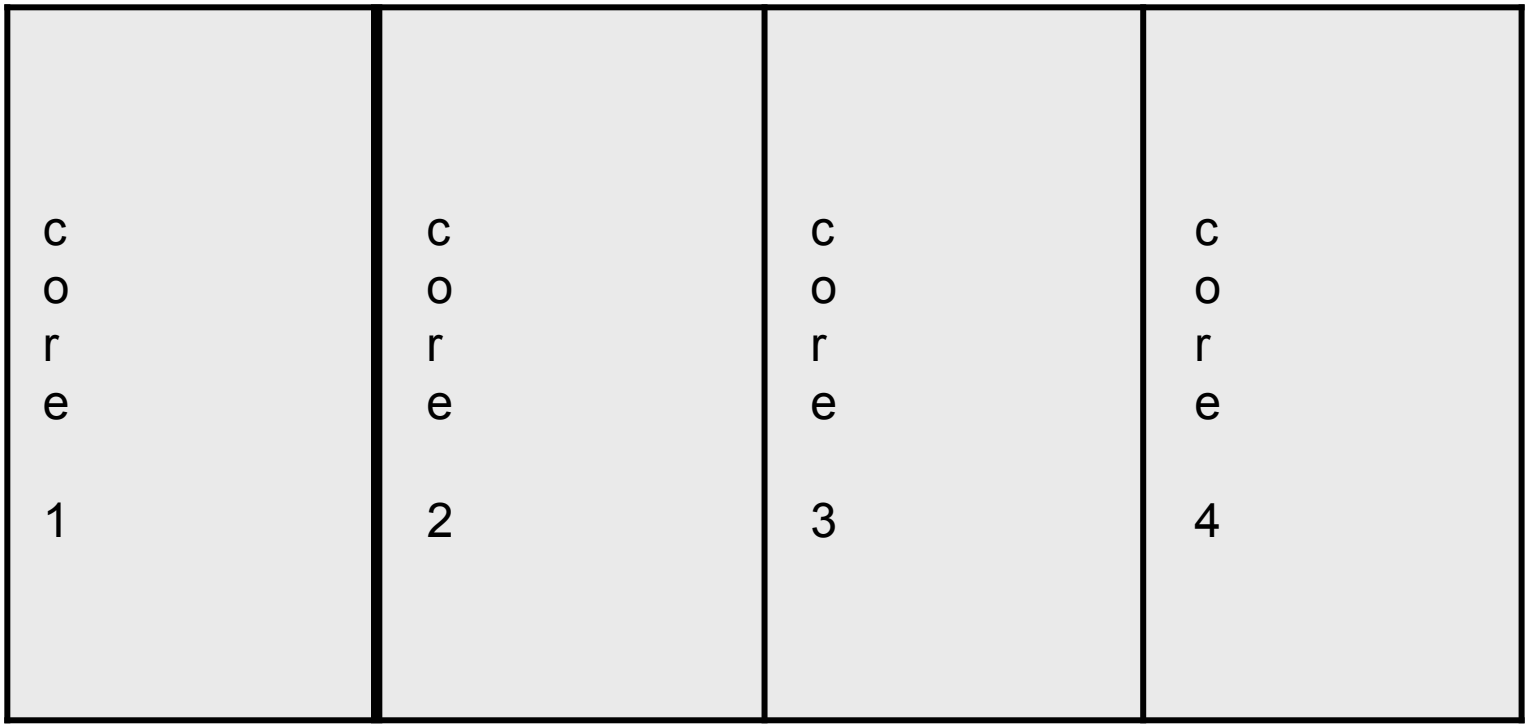
- Replicate multiple processor cores on a single die.

Core 1 Core 2 Core 3 Core 4

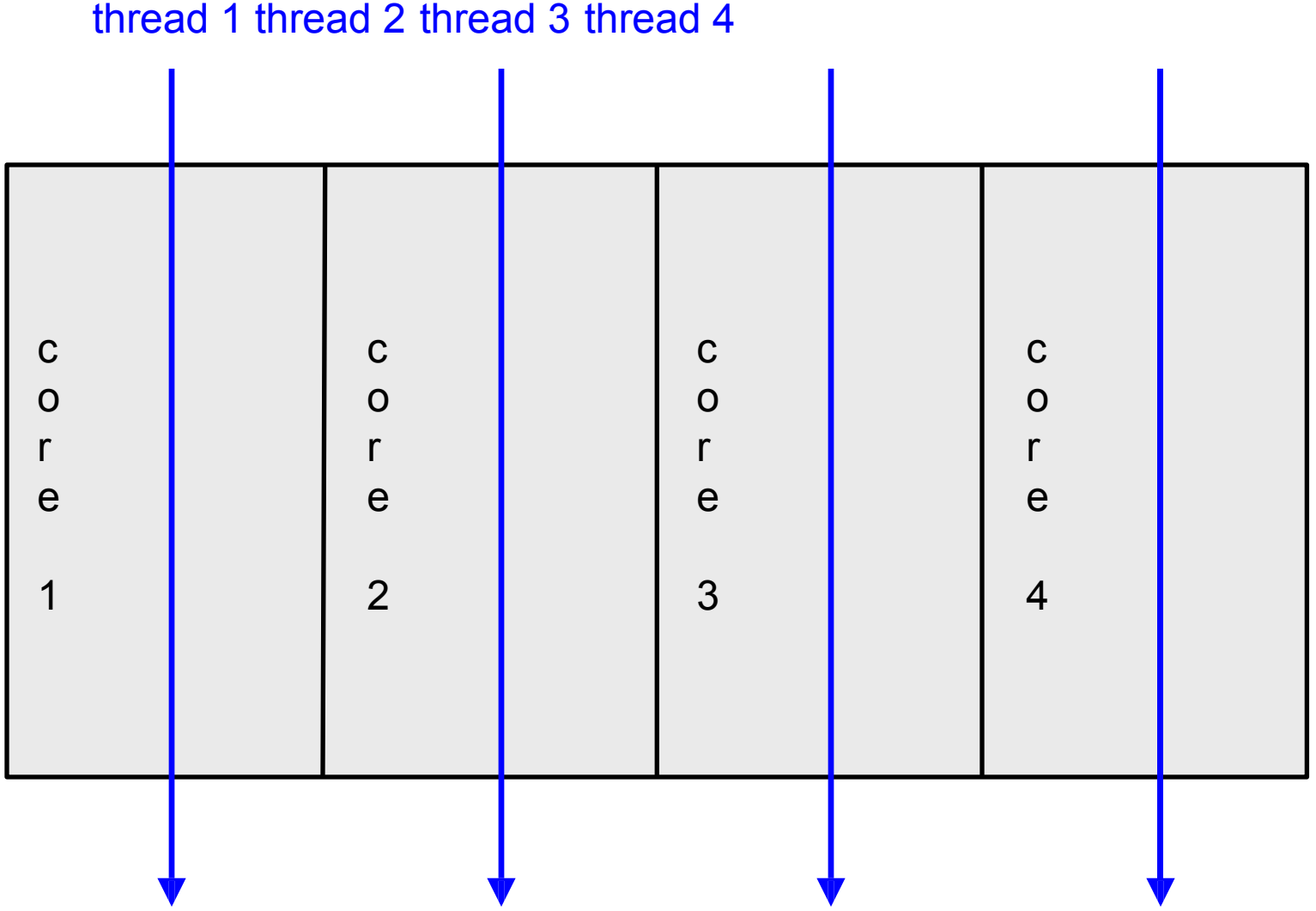


Multi-core CPU chip

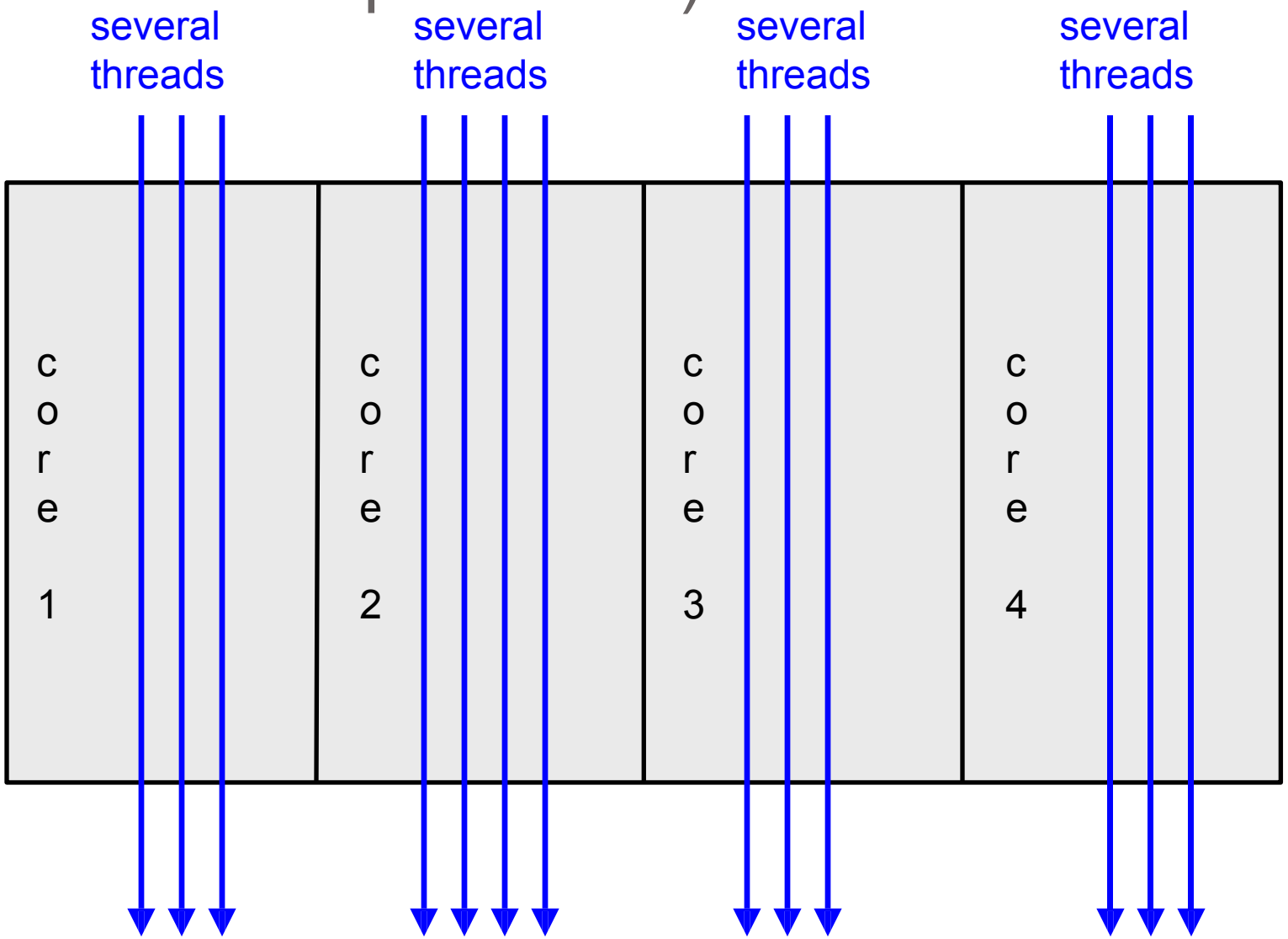
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



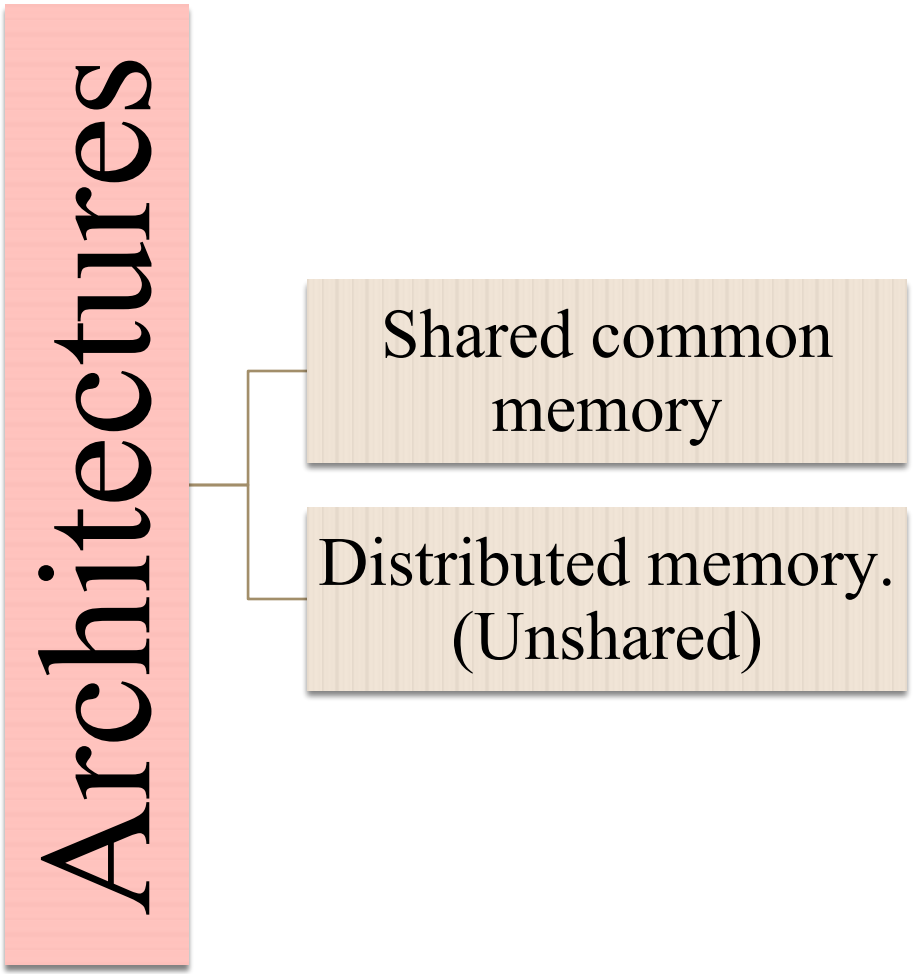
The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)

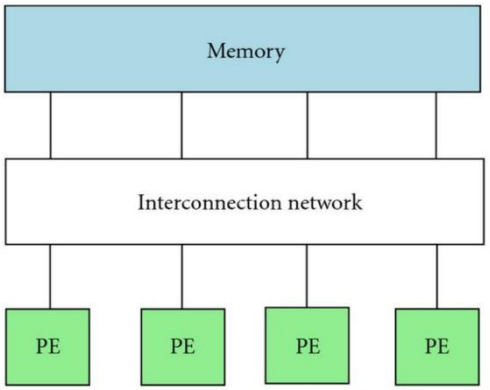


Memory in Multiprocessor System



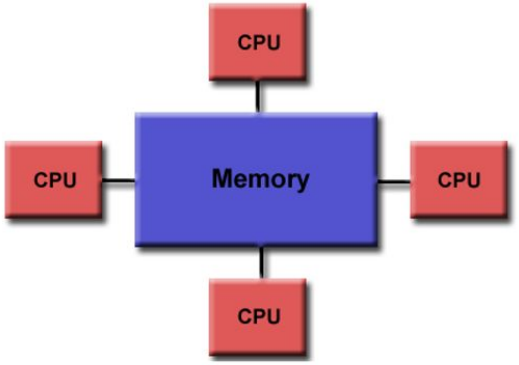
Shared Memory Multiprocessors

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: UMA , NUMA and COMA.



Uniform Memory Access (UMA)

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines.
- Identical processors .
- Equal access and access times to memory .
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.



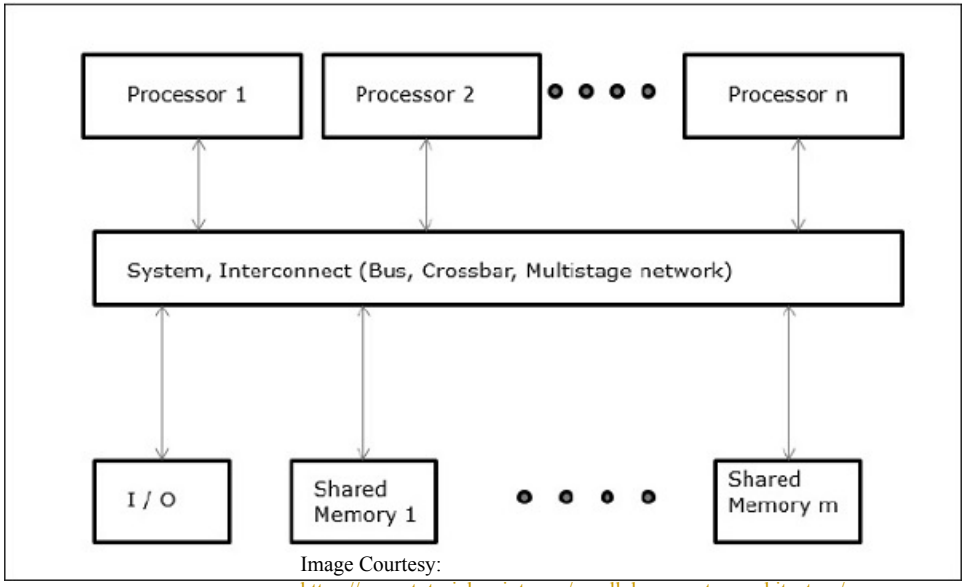
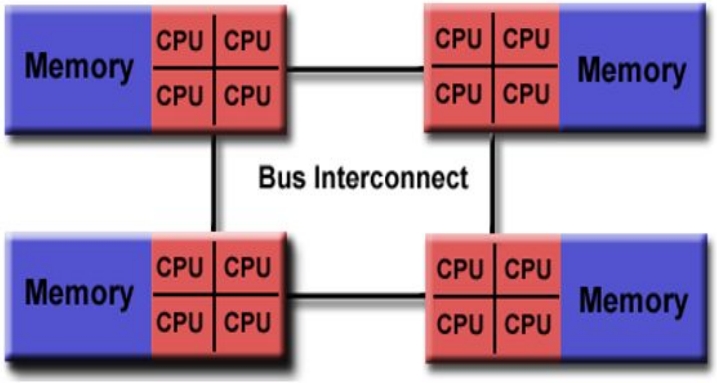


Image Courtesy:
https://www.tutorialspoint.com/parallel_computer_architecture/parallel_computer_architecture_models.htm

Non-Uniform Memory Access (NUMA)

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories .
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA



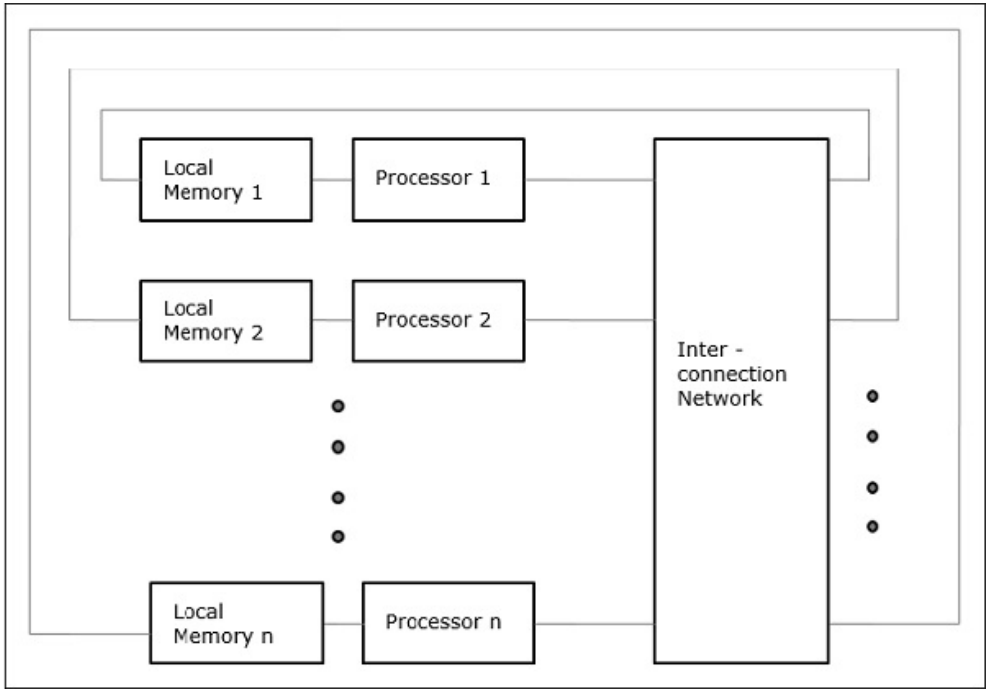


Image Courtesy:
https://www.tutorialspoint.com/parallel_computer_architecture/parallel_computer_architecture_models.htm

COMA Model

- It is a special case of NUMA machine in which the distributed main memories are converted to caches. All caches form a global address space and there is no memory hierarchy at each processor node.

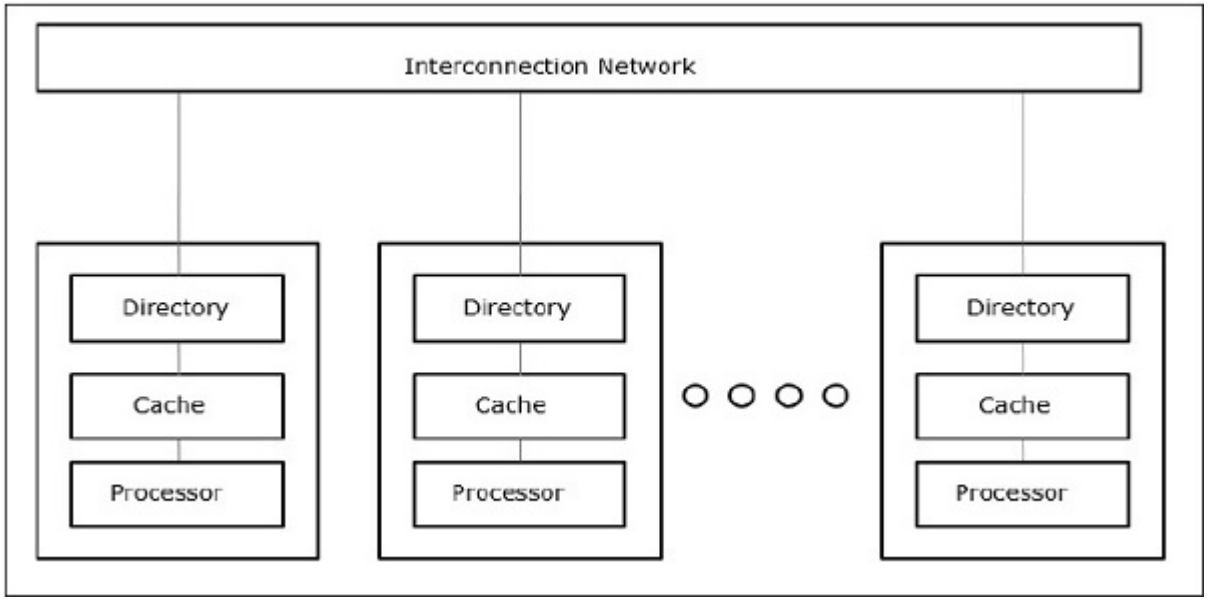


Image Courtesy:
https://www.tutorialspoint.com/parallel_computer_architecture/parallel_computer_architecture_models.htm

Distributed Memory Systems

- Distributed memory systems each processor has its own memory and it require a communication network to connect inter-processor memory.
- Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently.

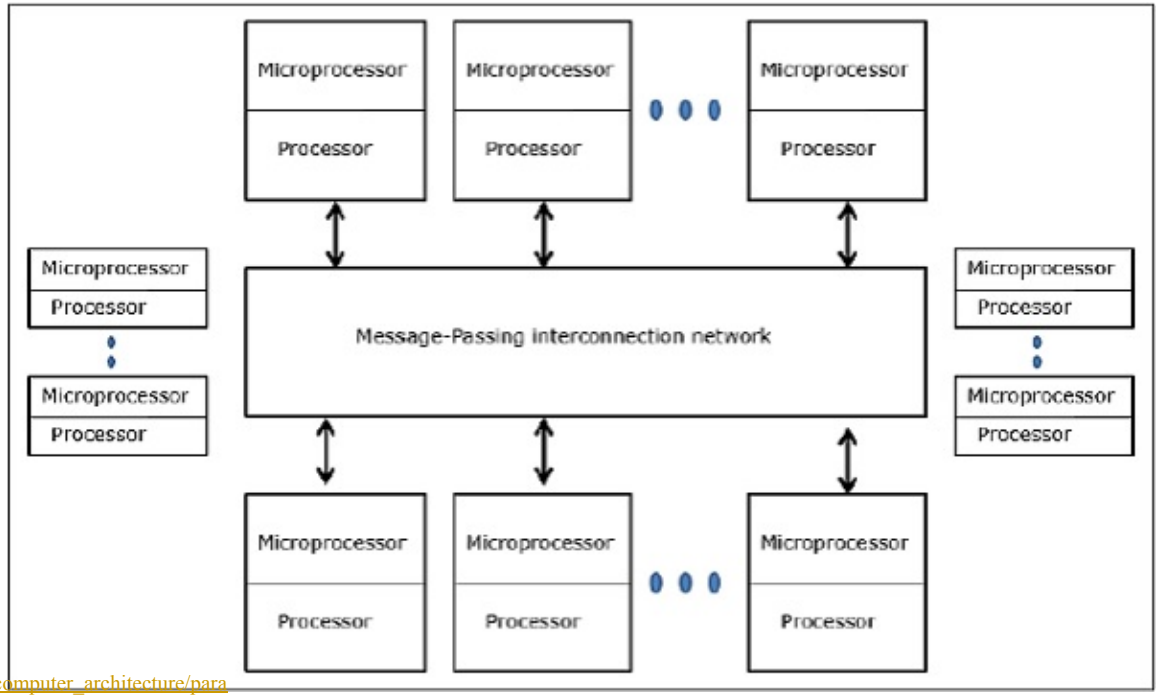


Image Courtesy:
https://www.tutorialspoint.com/parallel_computer_architecture/parallel_computer_architecture_models.htm

- Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.
- Synchronization between tasks is likewise the programmer's responsibility

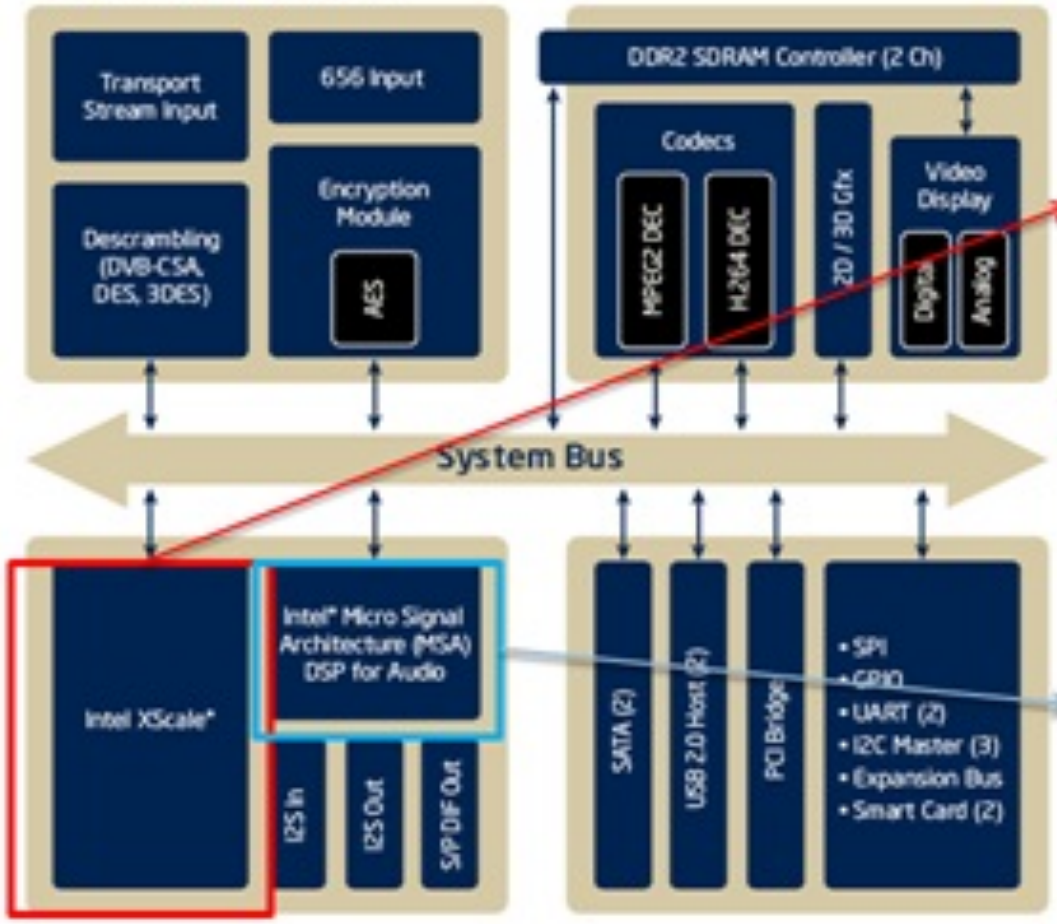
- Cache Coherency in Multiprocessor Systems

Examples of Multi-Core Processors

- Homogeneous Multi-Core Processor
  - Contains identical processor cores that support the same instruction set architecture (ISA)

- Heterogeneous Multi-Core Processor
  - Consist of non-identical processor cores that support different ISA
  - For example: Intel CE 2110 Media that consists of an Intel Xscale processor core and an Intel Micro Signal Architecture (MSA) DSP core.
  - Each processor usually has own specialty

Intel CE 2110 Media



Intel Xscale:

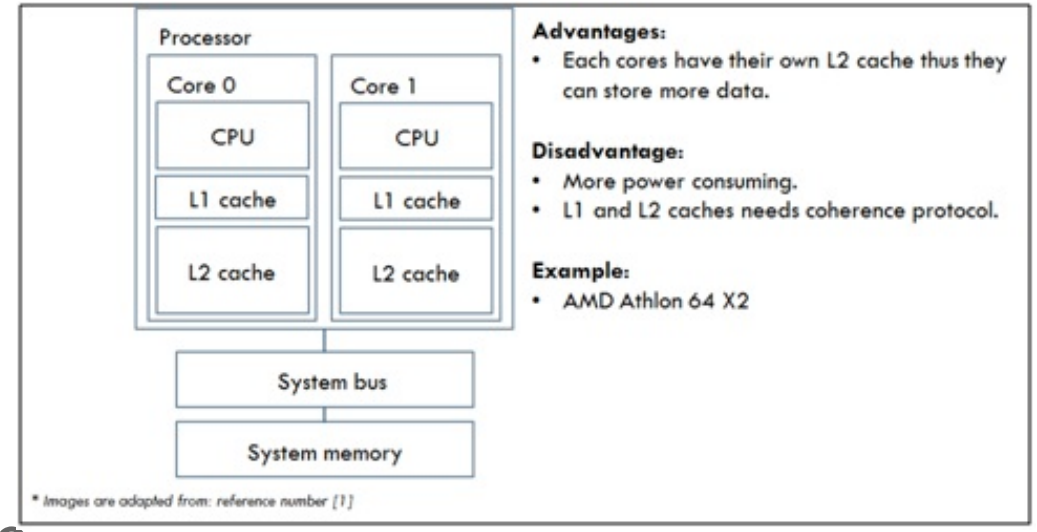
- General Tasks

Intel Micro Signal Architecture (MSA) [8]

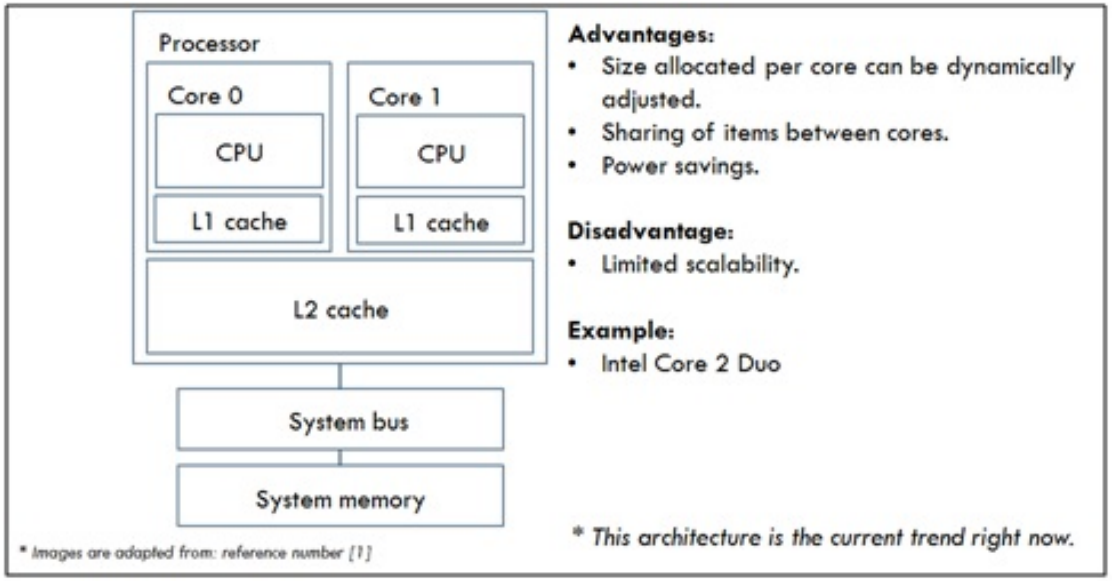
- Audio codecs
- PowerVR* 2D/3D graphics accelerator
- Hardware accelerators for encryption and decryption
- Comprehensive peripheral interfaces

* Images are taken from: www.linuxfordevices.com [7]

- Further it contains Separate L2 cache [1]



- Shared L2 cache



- **Advantages**

Having a multi-core processor in a computer means that it will work faster for certain programs.

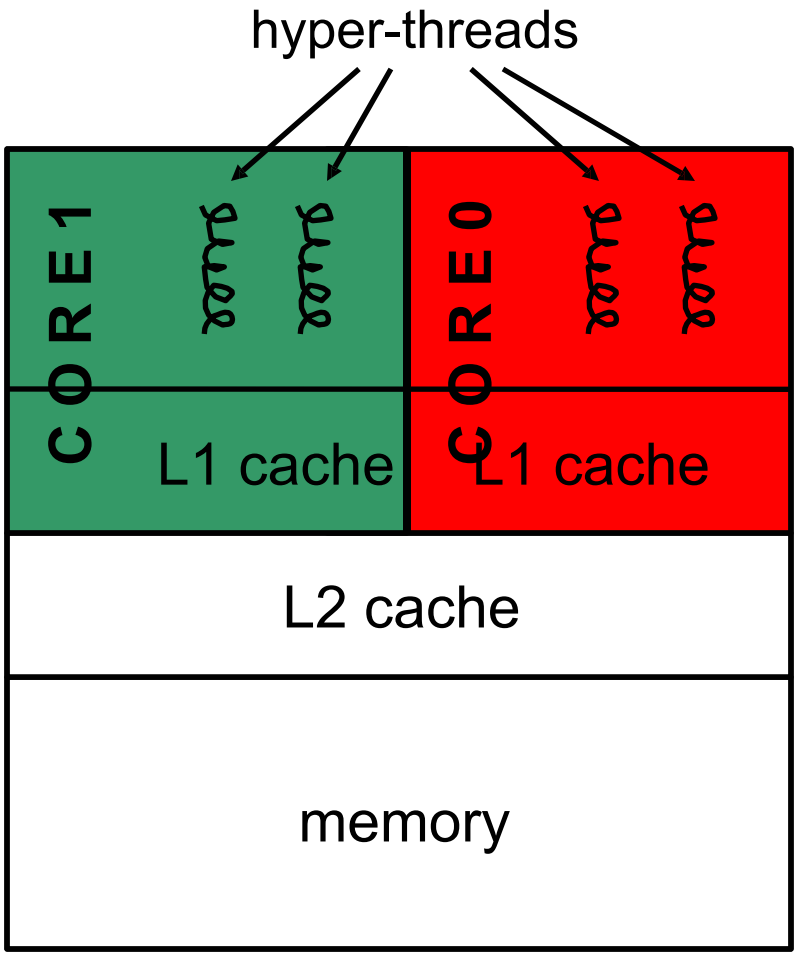
- The computer may not get as hot when it is turned on.
- The computer needs less power because it can turn off some sections if they aren´t needed.
- More features can be added to the computer.
- The signals between different CPUs travel shorter distances, therefore they degrade less.

- **Disadvantages**

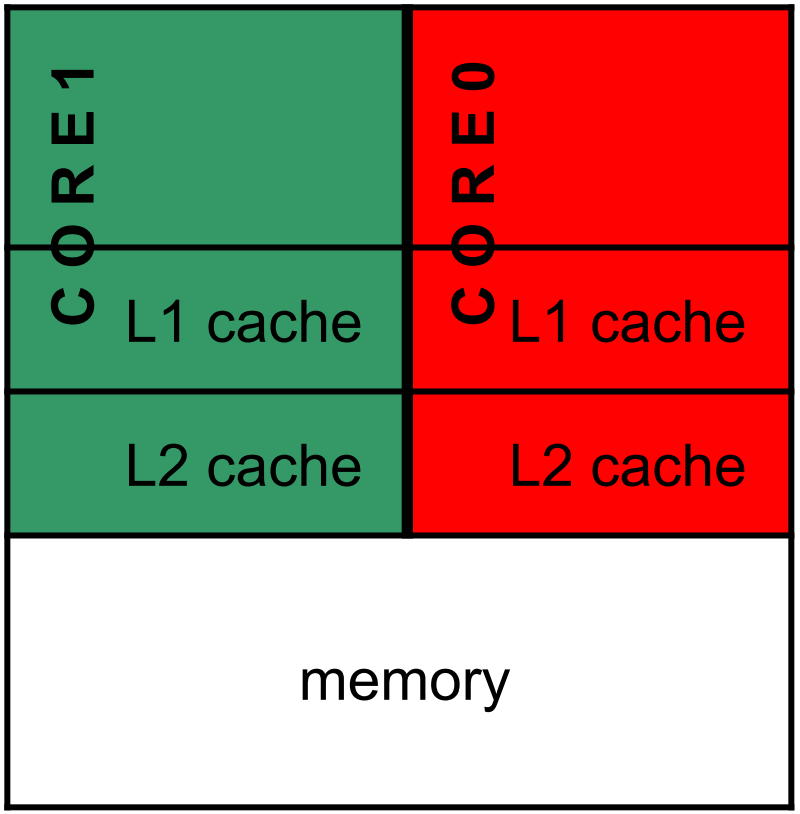
- They do not work at twice the speed as a normal processor. They get only 60-80% more speed.
- The speed that the computer works at depends on what the user is doing with it.
- They cost more than single core processors.
- They are more difficult to manage thermally than lower-density single-core processors.
- Not all operating systems support more than one core.
- Operating systems compiled for a multi-core processor will run slightly slower on a single-core processor.

“Fish” machines

- Dual-core
  Intel Xeon processors
- Each core is
  hyper-threaded
- Private L1 caches
- Shared L2 caches

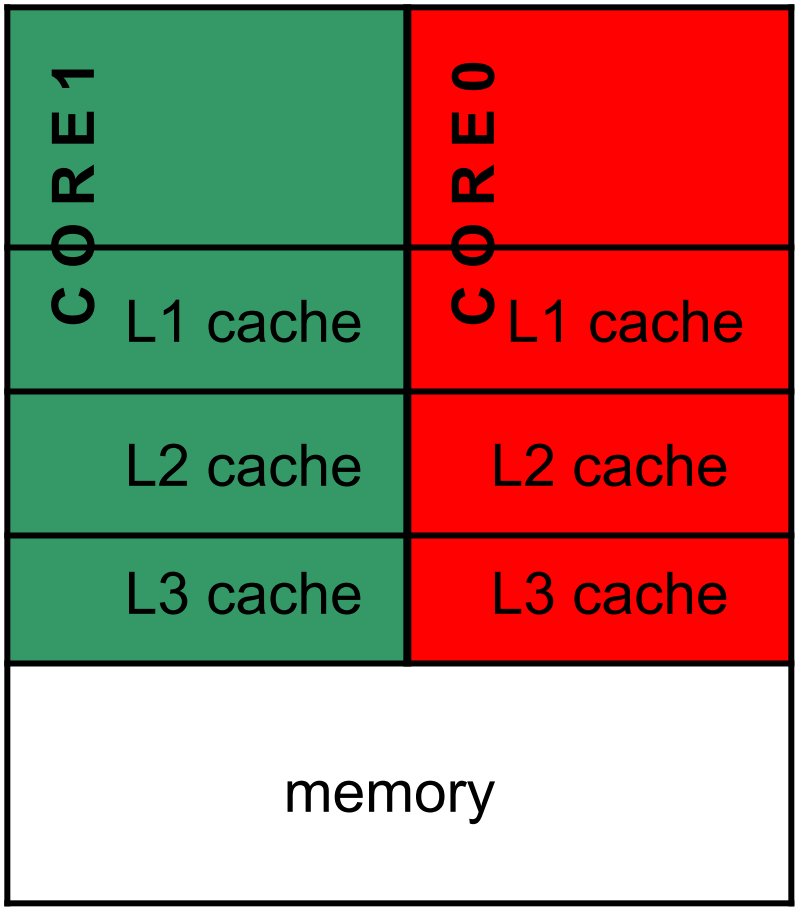


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D



A design with L3 caches

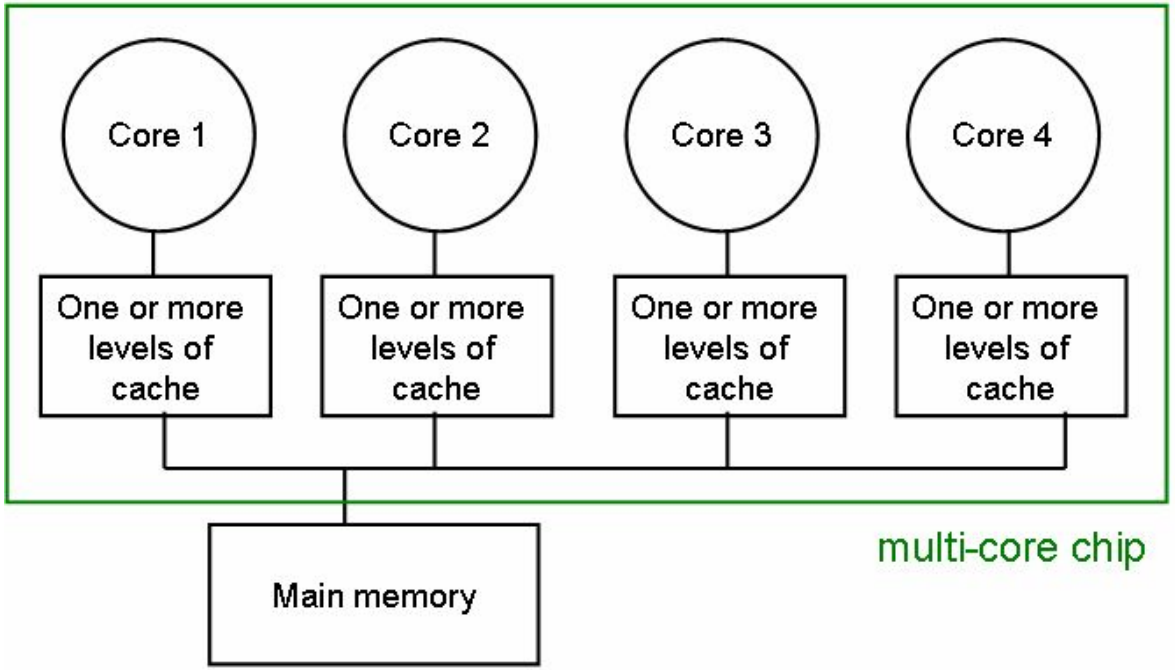
Example: Intel Itanium 2

Private vs Shared Caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

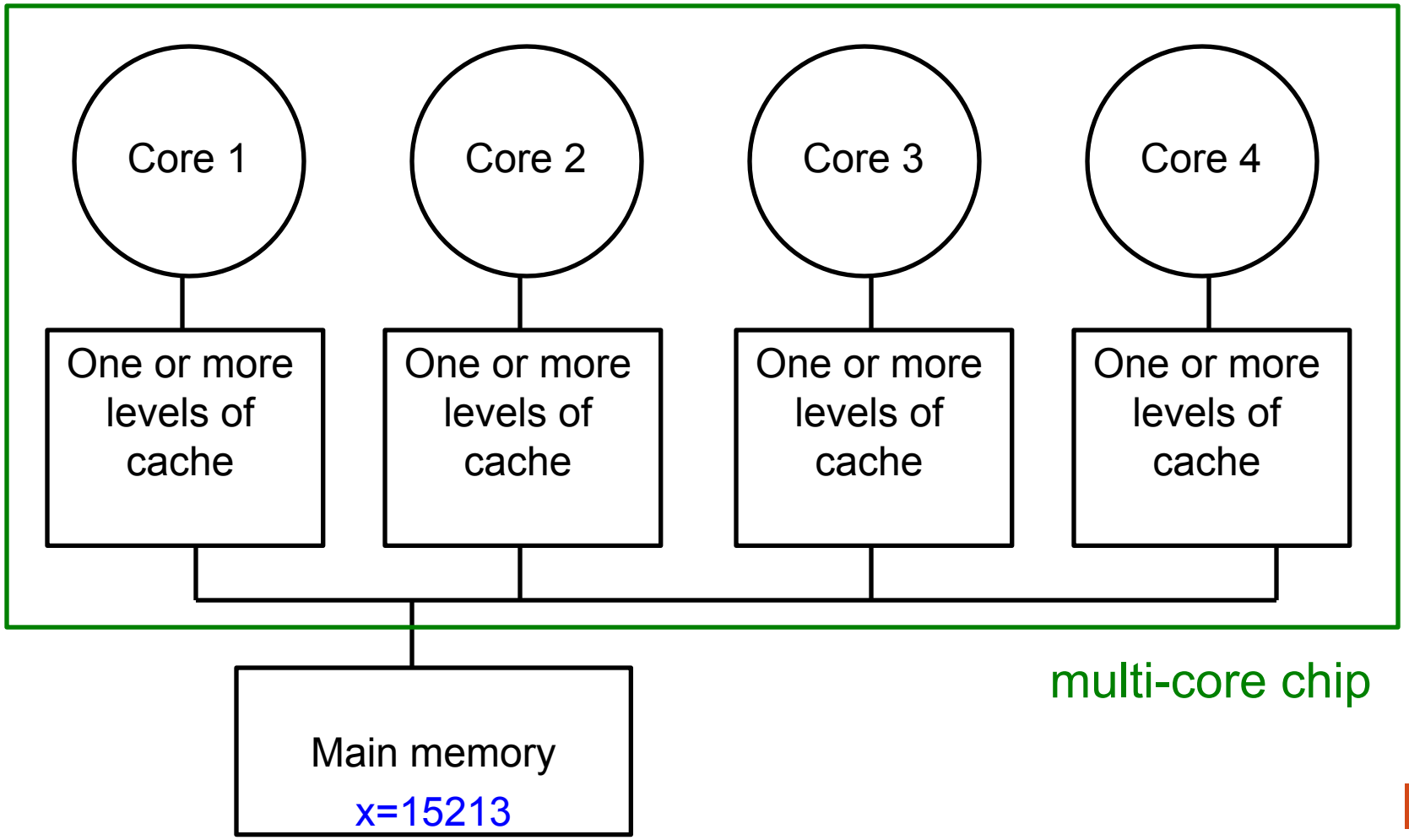
The cache coherence problem

- Since we have private caches:
  How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores



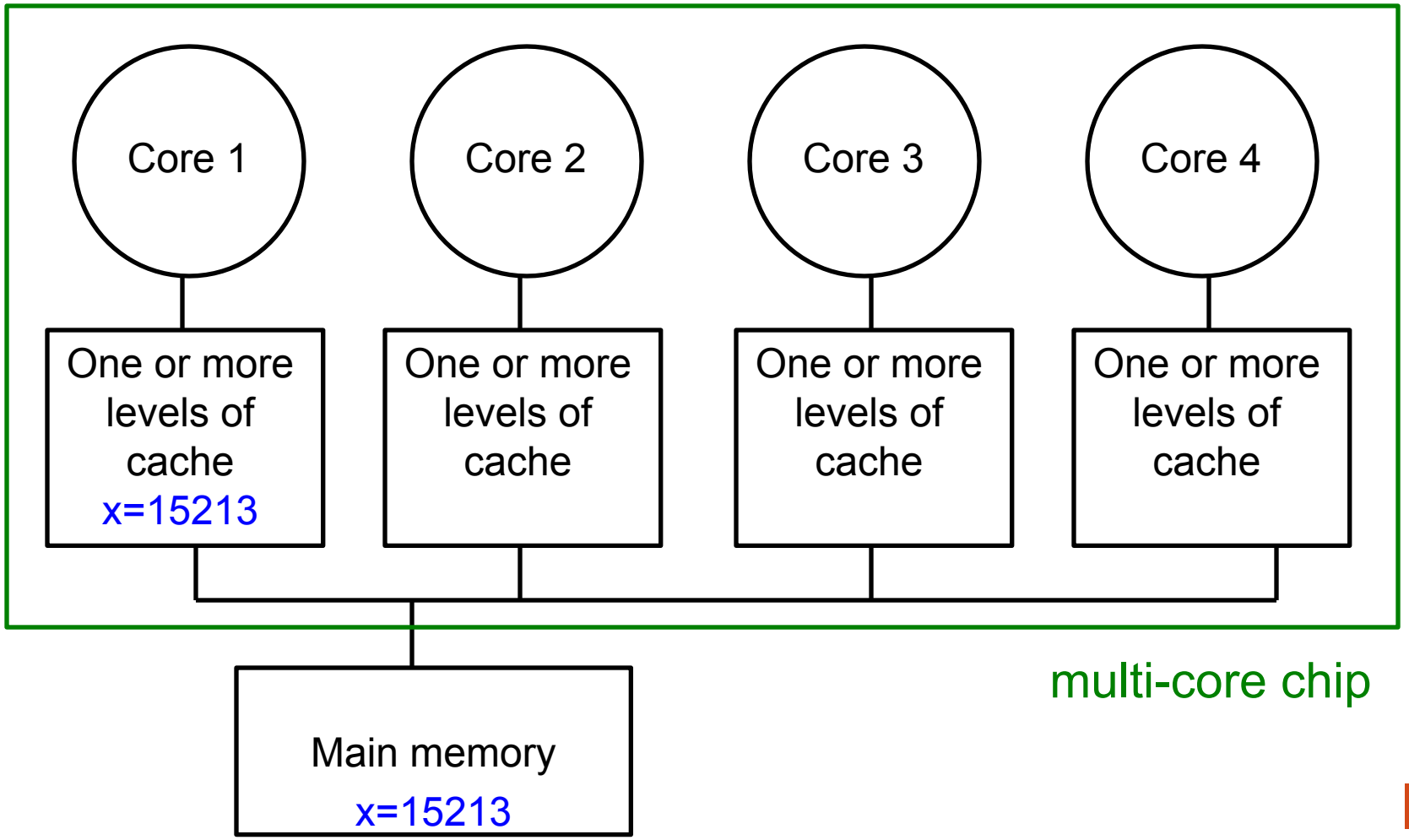
The cache coherence problem

Suppose variable x initially contains 15213



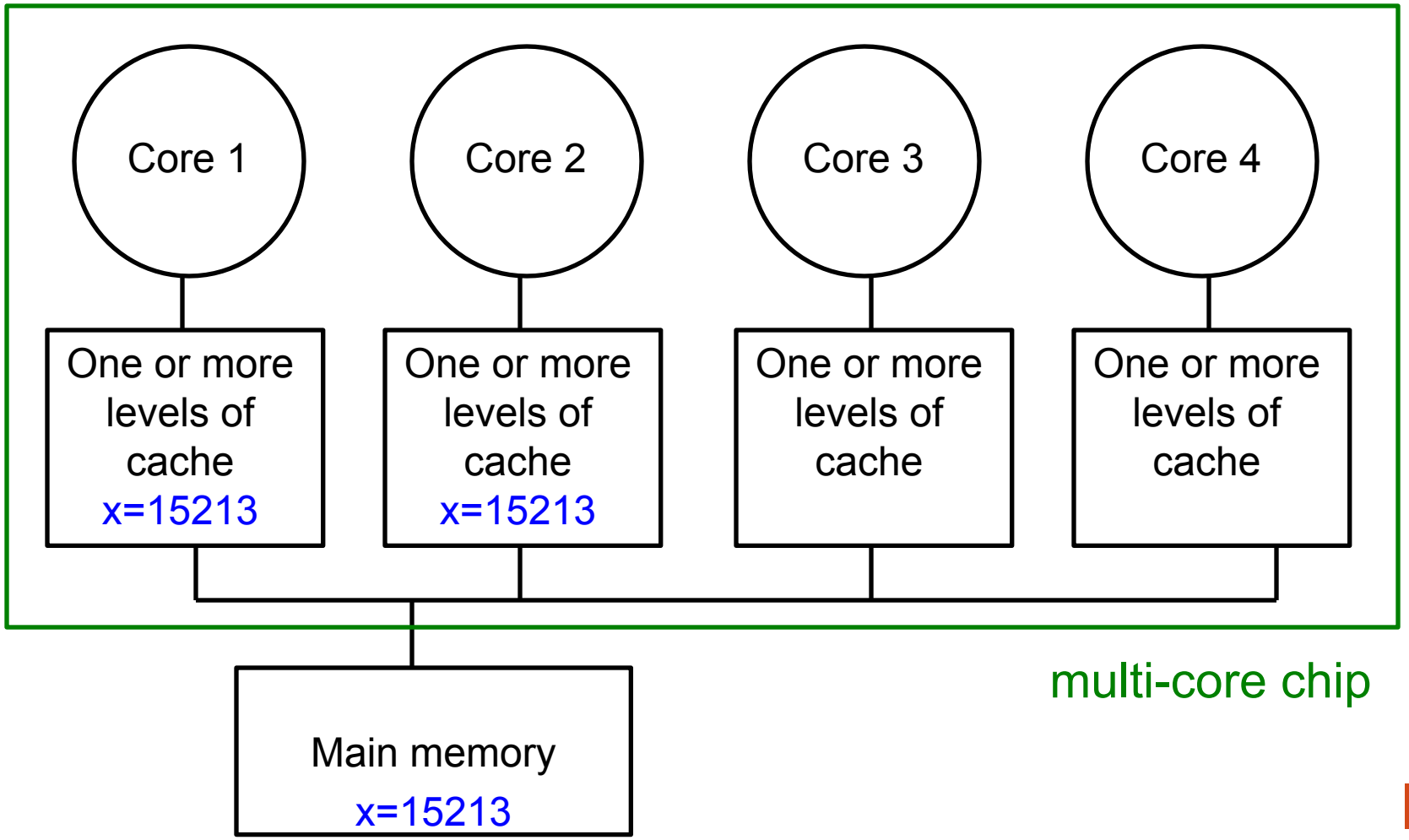
The cache coherence problem

Core 1 reads x



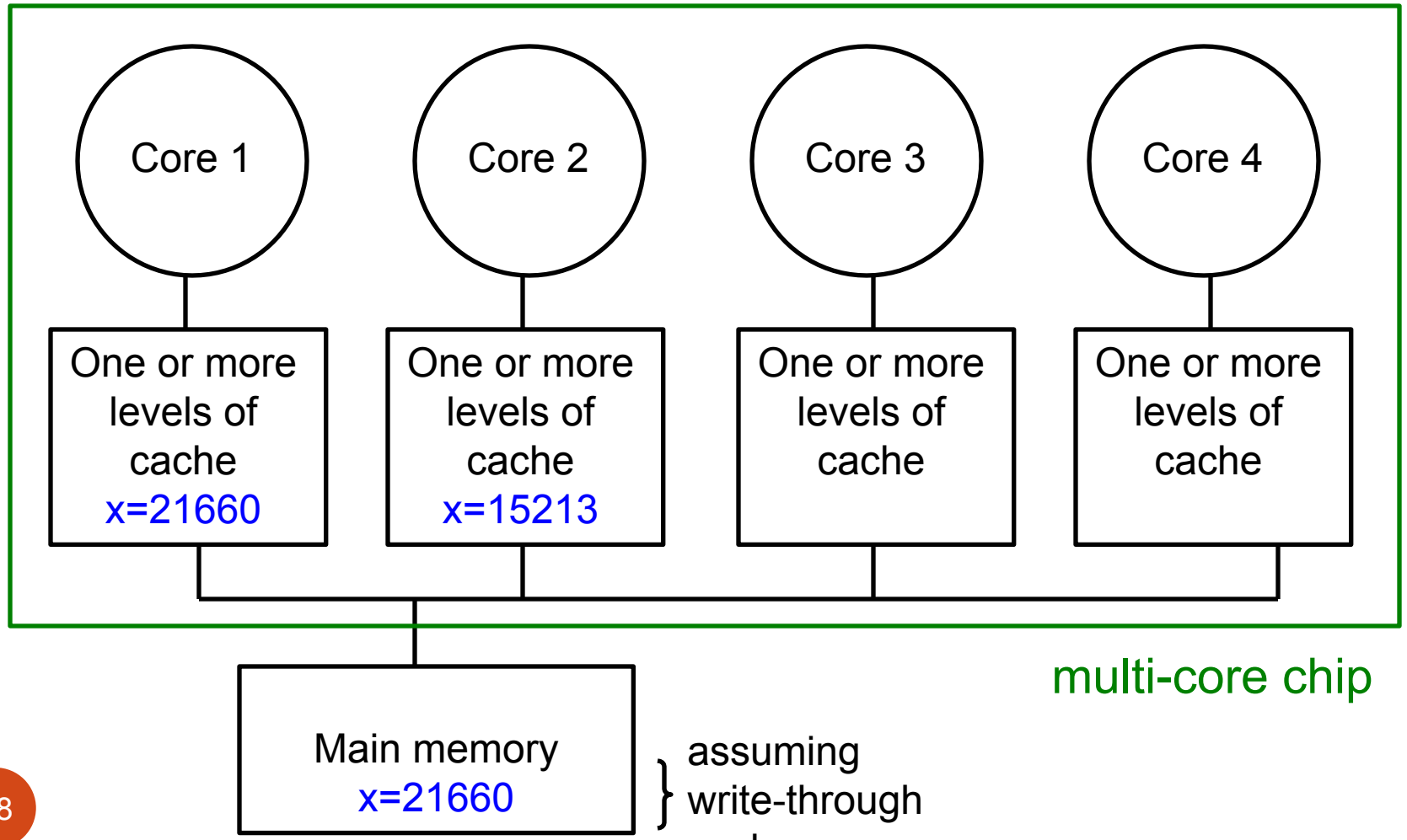
The cache coherence problem

Core 2 reads x



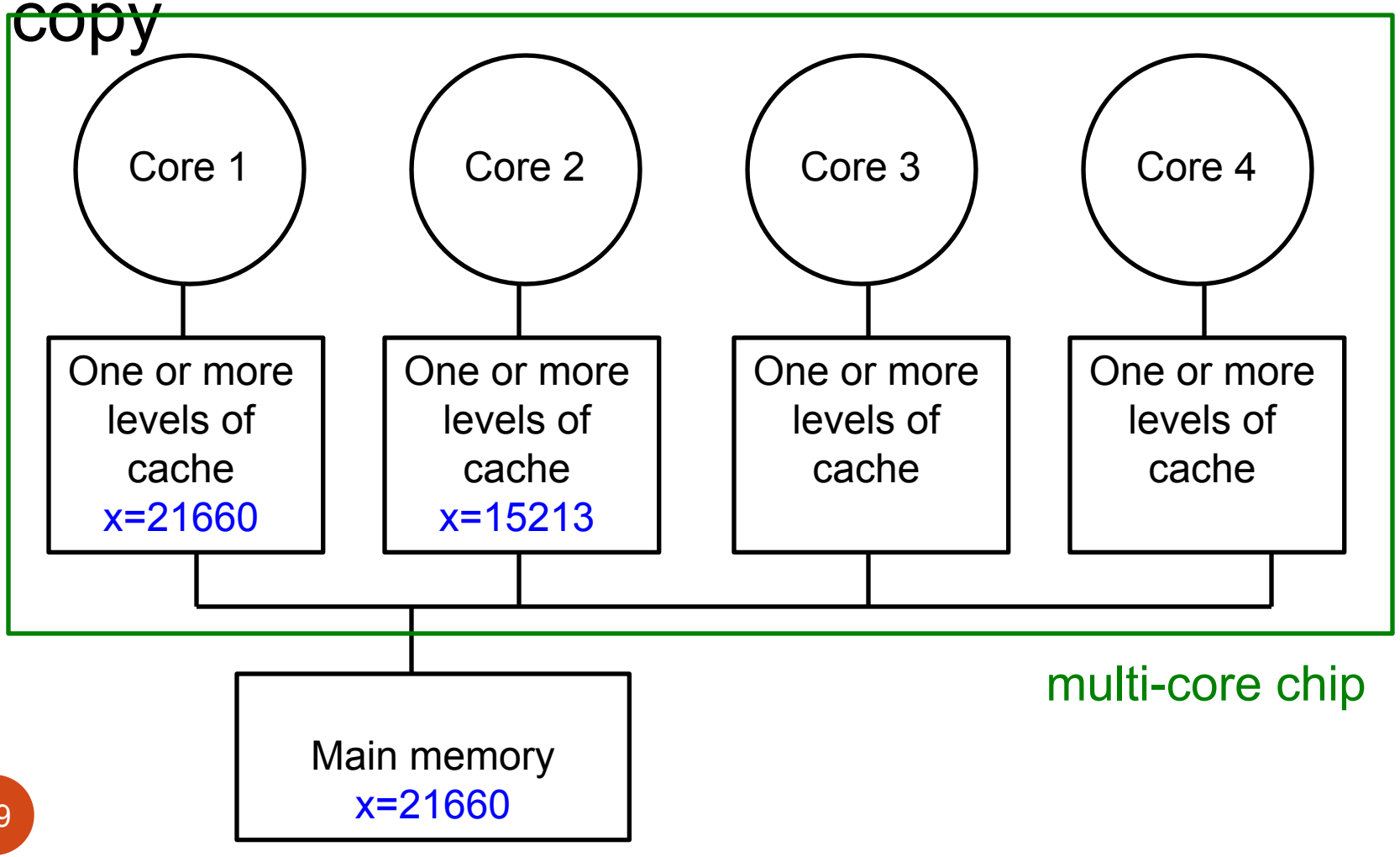
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

Core 2 attempts to read x… gets a stale copy

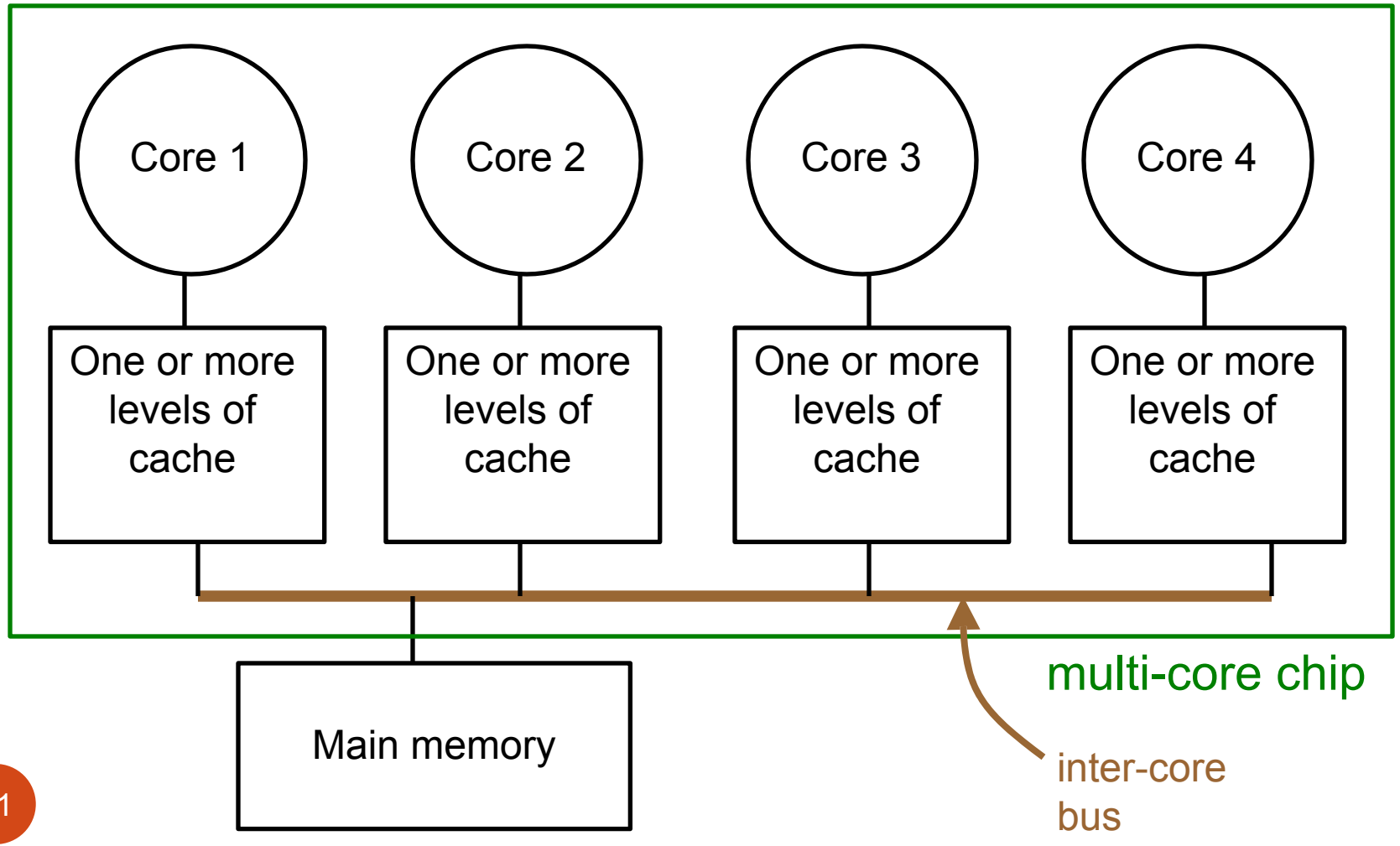


Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.

- A simple solution: *invalidation*-based protocol with *snooping*

Inter-core bus

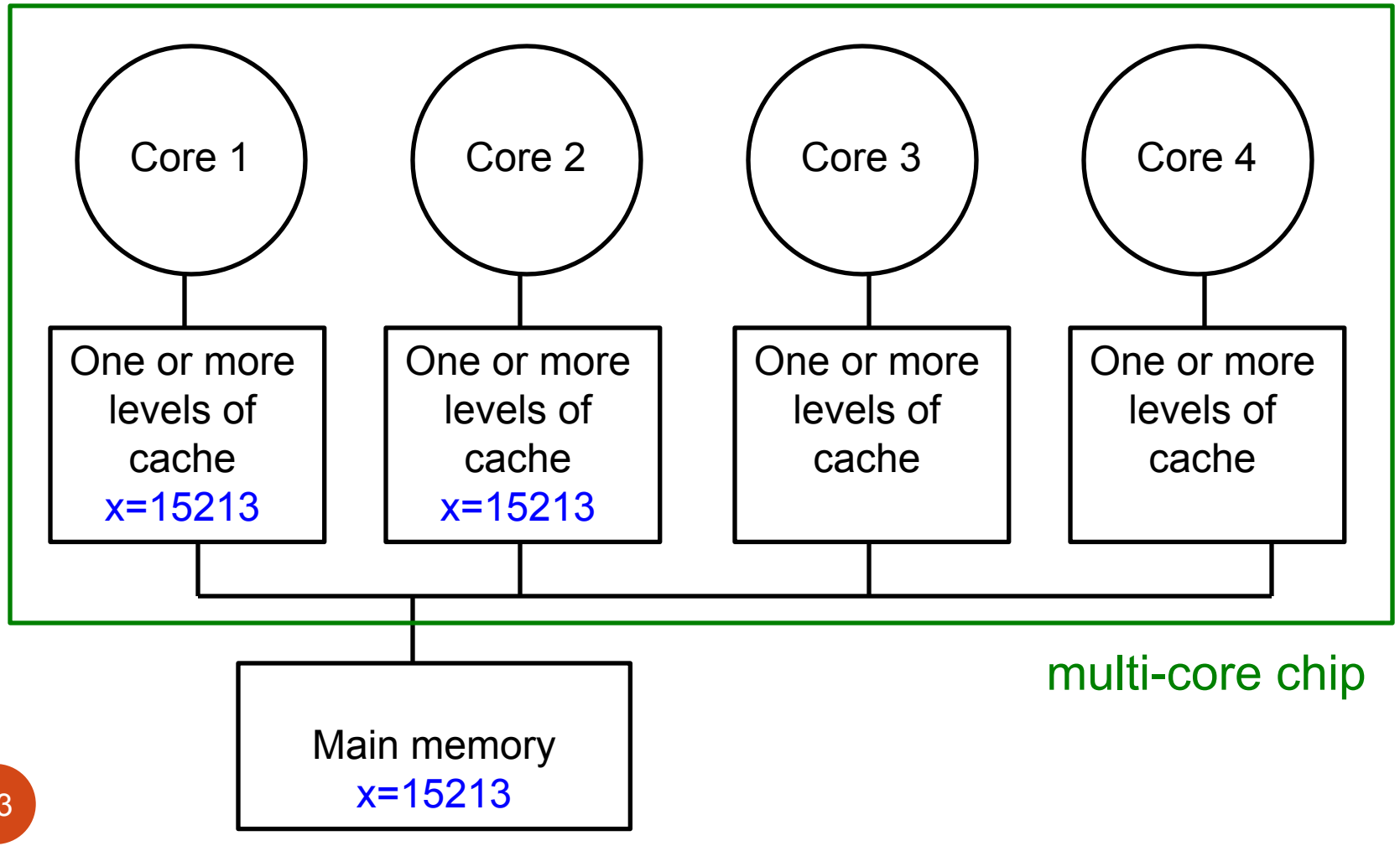


Invalidation protocol with snooping

- Invalidation:
  If a core writes to a data item, all other copies of this data item in other caches are *invalidated*
- Snooping:
  All cores continuously "snoop" (monitor) the bus connecting the cores.

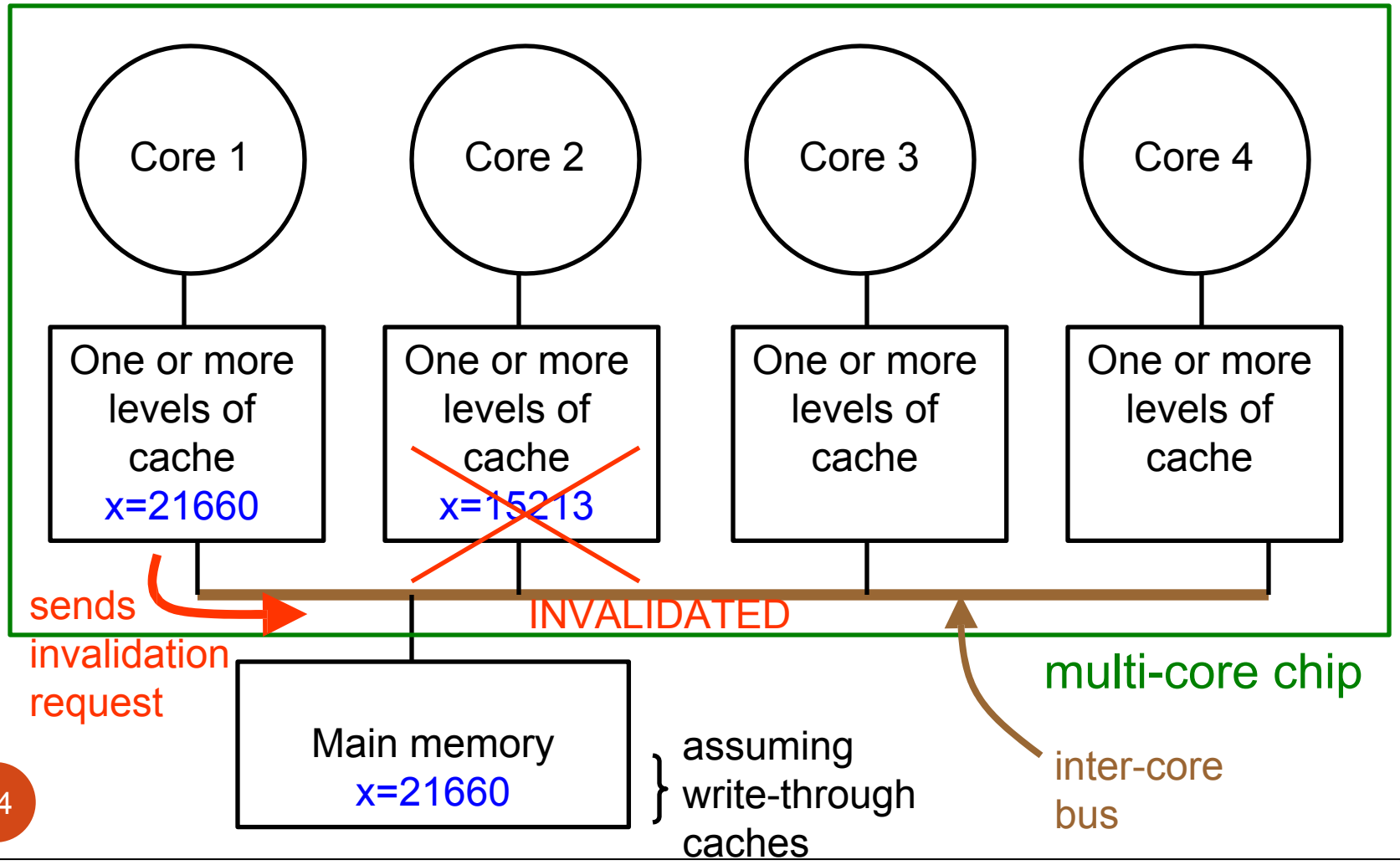
The cache coherence problem

Revisited: Cores 1 and 2 have both read x



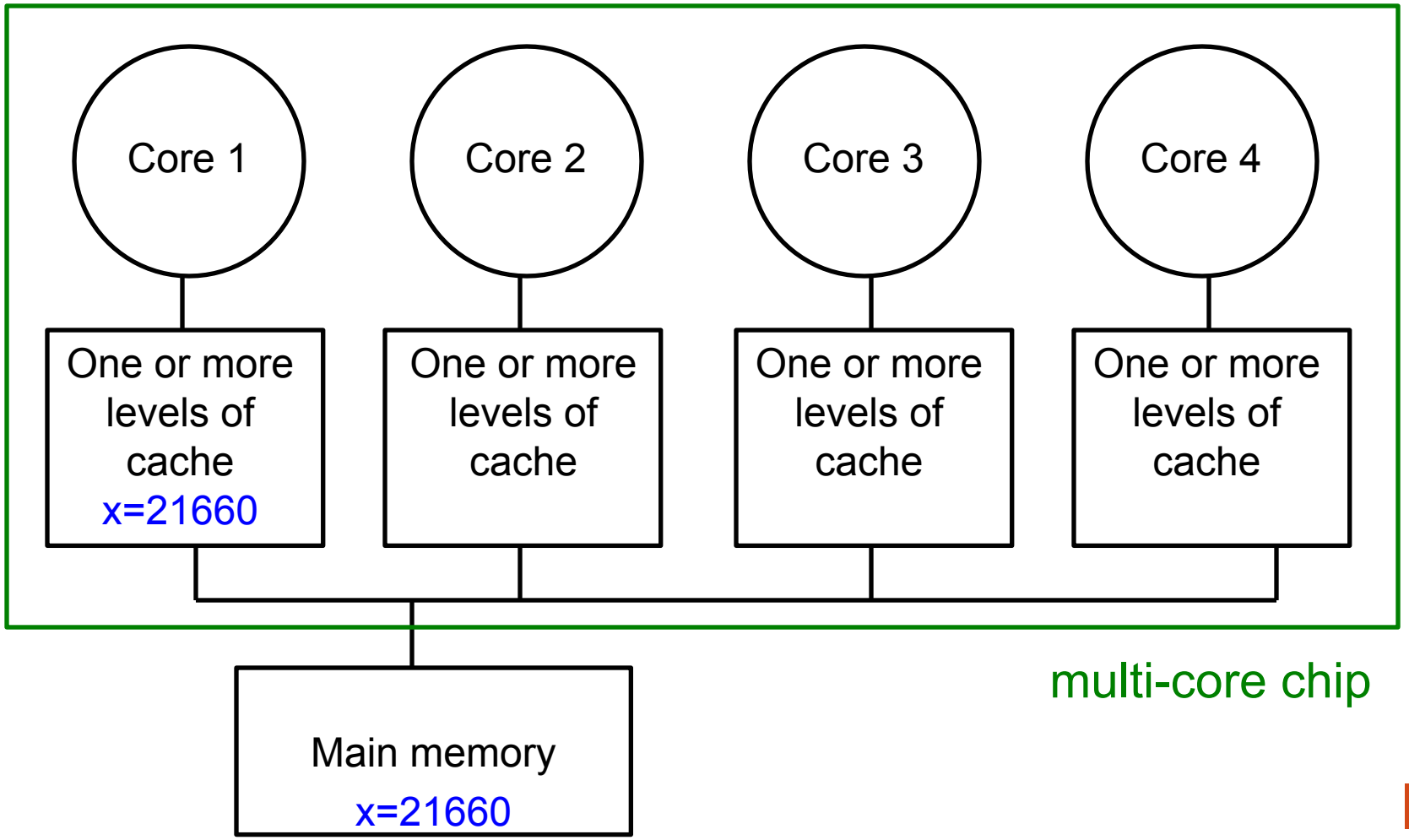
The cache coherence problem

Core 1 writes to x, setting it to 21660



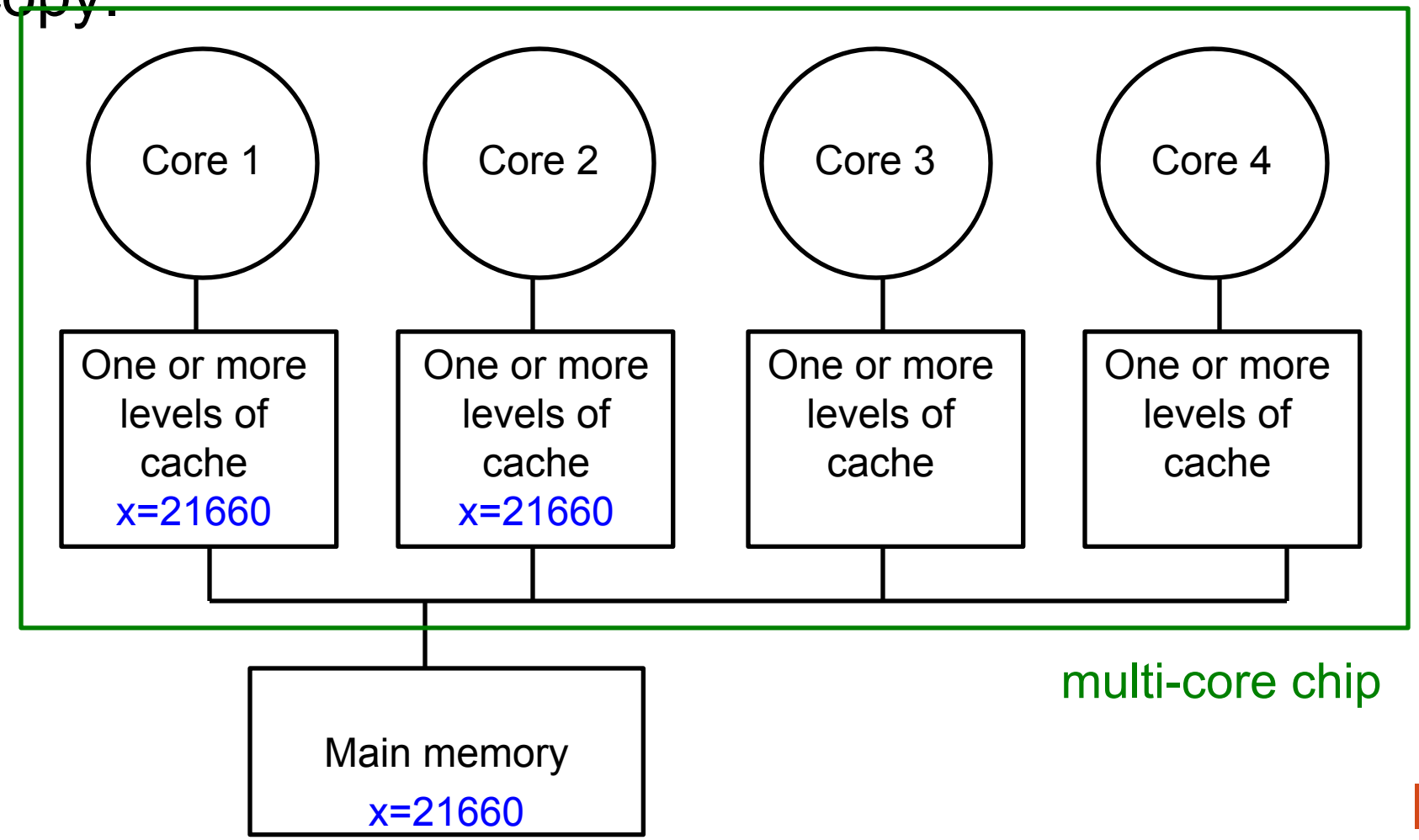
The cache coherence problem

After invalidation:



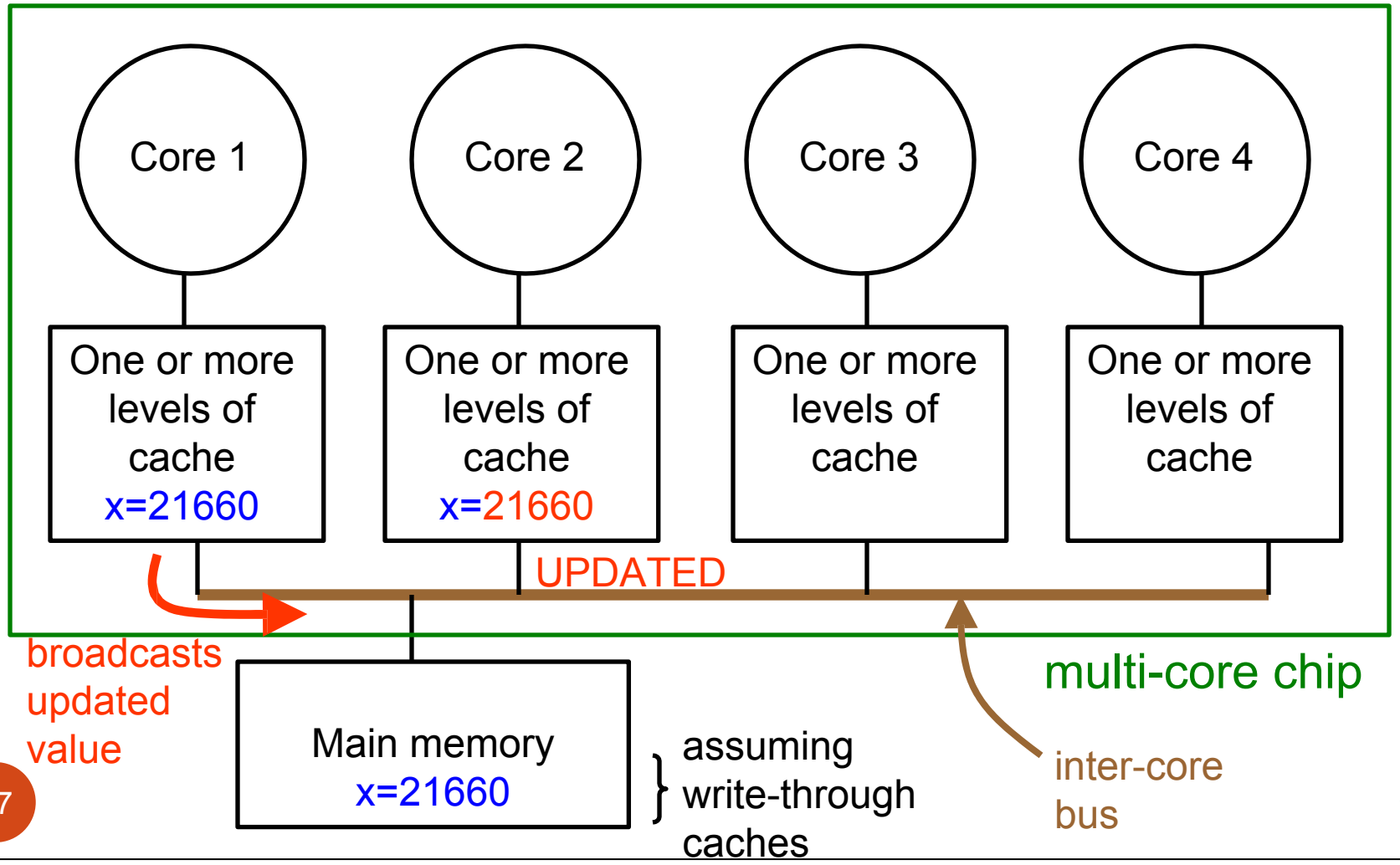
The cache coherence problem

Core 2 reads x. Cache misses, and loads the new copy.



Alternative to invalidate protocol: update protocol

Core 1 writes x=21660:



Invalidation vs update

- Multiple writes to the same location
  - invalidation: only the first time
  - update: must broadcast each write
    (which includes new variable value)

- Invalidation generally performs better: it generates less bus traffic

Invalidation protocols

- This was just the basic invalidation protocol
- More sophisticated protocols use extra cache state bits
- MSI, MESI

  (Modified, Exclusive, Shared, Invalid)

MESI Protocol

For Multiprocessor Systems

- The MESI protocol is one implementation of enforcing data integrity among caches sharing data; the write once write policy is a method to keep the protocol performing efficiently by avoiding superfluous data traffic on the system bus

MESI Protocol (1)

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage.
- Allows usage of a ‘write back’ scheme - i.e. main memory not updated until ‘dirty’ cache line is displaced
- Extension of usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal write back cache.

MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor 'dirty')
- **Exclusive** - cache line is the same as main memory and is the only cached copy
- **Shared** - Same as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (as in simple cache)

MESI Protocol (3)

- Cache line changes state as a function of memory access events.
- Event may be either
  - Due to local processor activity (i.e. cache access)
  - Due to bus activity - as a result of snooping
- Cache line has its own state affected only if address matches

MESI Protocol (4)

- Operation can be described informally by looking at action in local processor
  - Read Hit
  - Read Miss
  - Write Hit
  - Write Miss
- More formally by state transition diagram

MESI Local Read Hit

- Line must be in one of MES
- This must be correct local value (if M it must have been modified locally)
- Simply return value
- No state change

MESI Local Read Miss (1)

- No other copy in caches
  - Processor makes bus request to memory
  - Value read to local cache, marked E
- One cache has E copy
  - Processor makes bus request to memory
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Both lines set to S

MESI Local Read Miss (2)

- Several caches have S copy
  - Processor makes bus request to memory
  - One cache puts copy value on the bus (arbitrated)
  - Memory access is abandoned
  - Local processor caches value
  - Local copy set to S
  - Other copies remain S

MESI Local Read Miss (3)

- One cache has M copy
  - Processor makes bus request to memory
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Local copy tagged S
  - **Source (M) value copied back to memory**
  - Source value M -> S

MESI Local Write Hit (1)

Line must be one of MES

- M
  - line is exclusive and already ‘dirty’
  - Update local cache value
  - no state change
- E
  - Update local cache value
  - State E -> M

MESI Local Write Hit (2)

- S
  - Processor broadcasts an invalidate on bus
  - Snooping processors with S copy change S->I
  - Local cache value is updated
  - Local state change S->M

MESI Local Write Miss (1)

Detailed action depends on copies in other processors

- No other copies
  - Value read from memory to local cache (?)
  - Value updated
  - Local copy state set to M

MESI Local Write Miss (2)

- Other copies, either one in state E or more in state S
- Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
- Snooping processors see this and set their copy state to I
- Local copy updated & state set to M

MESI Local Write Miss (3)

Another copy in state M

- Processor issues bus transaction marked RWITM
- Snooping processor sees this
  - Blocks RWITM request
  - Takes control of bus
  - Writes back its copy to memory
  - Sets its copy state to I

MESI Local Write Miss (4)

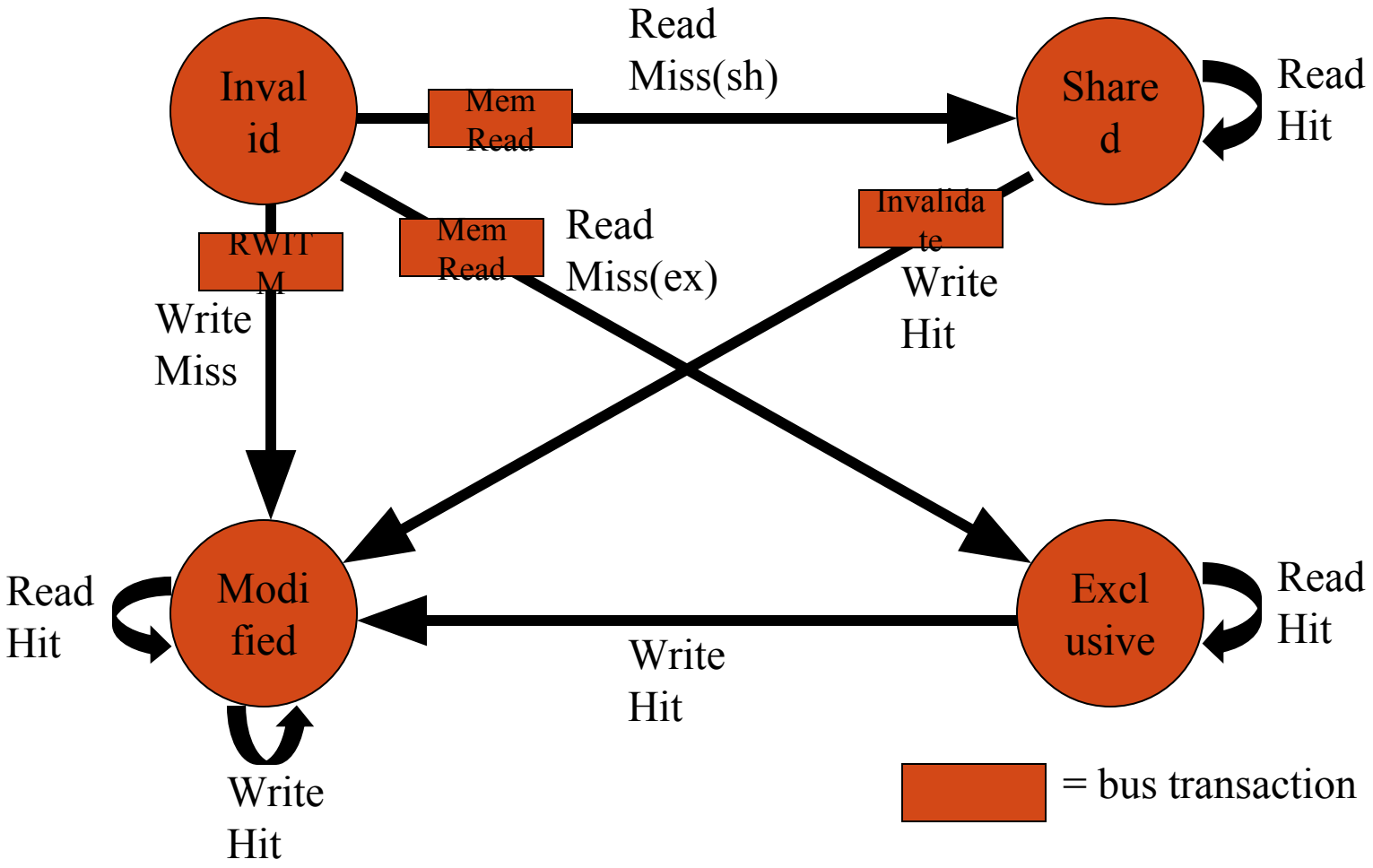
Another copy in state M (continued)

- Original local processor re-issues RWITM request
- Is now simple no-copy case
  - Value read from memory to local cache
  - Local copy value updated
  - Local copy state set to M

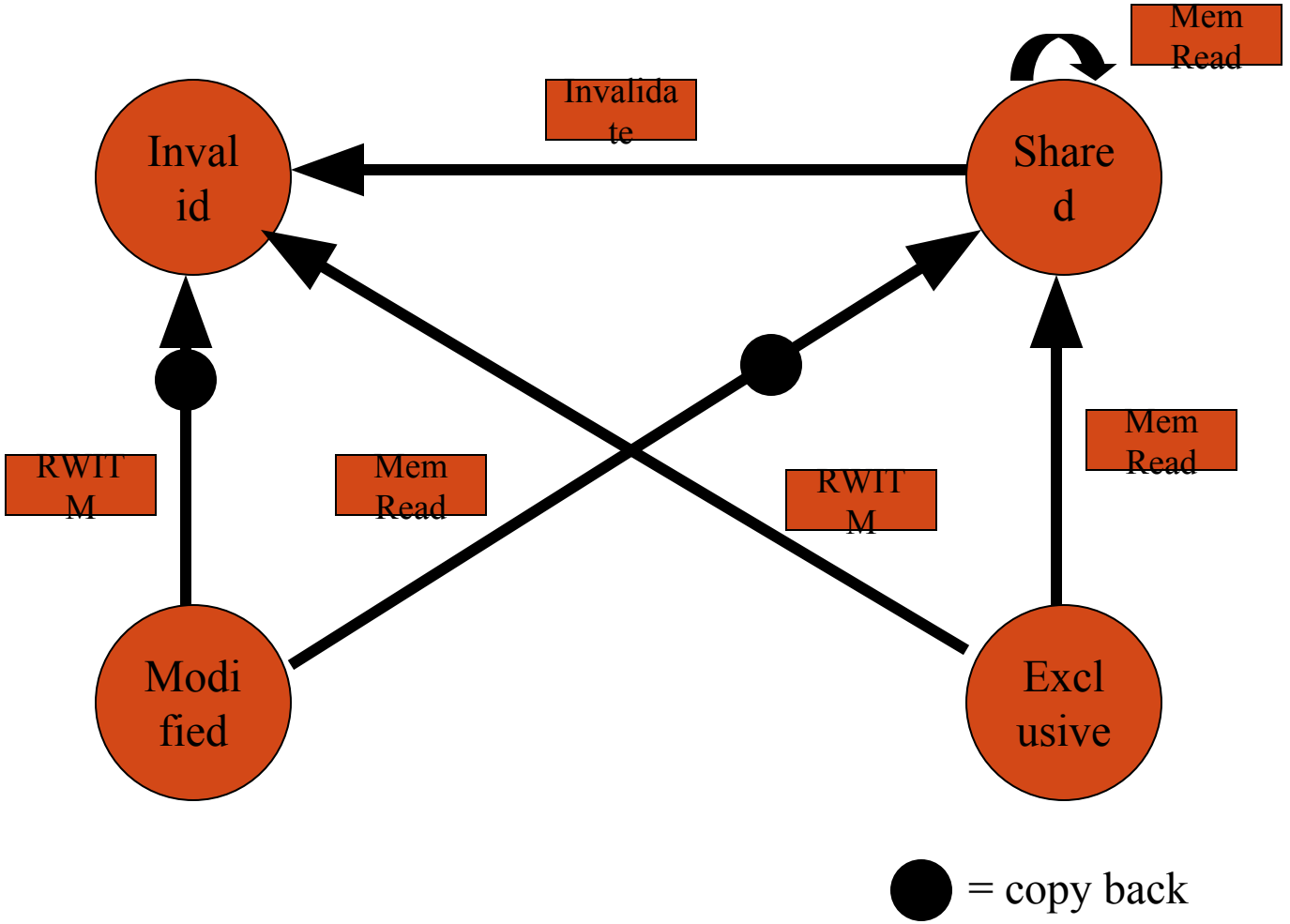
Putting it all together

- All of this information can be described compactly using a state transition diagram
- Diagram shows what happens to a cache line in a processor as a result of
  - memory accesses made by that processor (read hit/miss, write hit/miss)
  - memory accesses made by other processors that result in bus transactions observed by this snoopy cache (Mem read, RWITM,Invalidate)

MESI – locally initiated accesses



MESI – remotely initiated accesses



MESI notes

- There are minor variations (particularly to do with write miss)
- Normal ‘write back’ when cache line is evicted is done if line state is M
- Multi-level caches
  - If caches are inclusive, only the lowest level cache needs to snoop on the bus

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
—Deemed to be University—

THANK YOU