Vel Tech

Rangarajan Dr. Sagunthala R&D Institute of Science and Technology

(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

Department of Computer Science and Engineering

COURSE CODE – TITLE: 1151CS110 – COMPUTER ORGANIZATION AND ARCHITECTURE

Course Instructor

Dr. M. Rajeev Kumar

Associate Professor (CSE)

UNIT IV- Parallelism

CO Nos.	Course Outcome(s)	Level of learning domain (Based on revised Bloom’s)
CO4	Understand parallel organization’s as advanced computer architectures and the working principles of multiprocessor.	K2

Instruction-level parallelism

Concepts and Challenges

The potential overlap among instructions is called **instruction-level parallelism** (**ILP**).

- ILP is a measure of the number of instructions that can be performed during a single clock cycle.
- Parallel instructions are a set of instructions that do not depend on each other to be executed.

Making Computers Think Parallel

Human

Write code yourself, directly controlling each processor

Compiler

Let the compiler convert your sequential code into parallel instructions.

Hardware

Concepts and Challenges *(Contd.,)*

Two approaches exploiting ILP:

- Hardware discovers and exploit the parallelism dynamically.
- Software finds parallelism, statically at compile time.

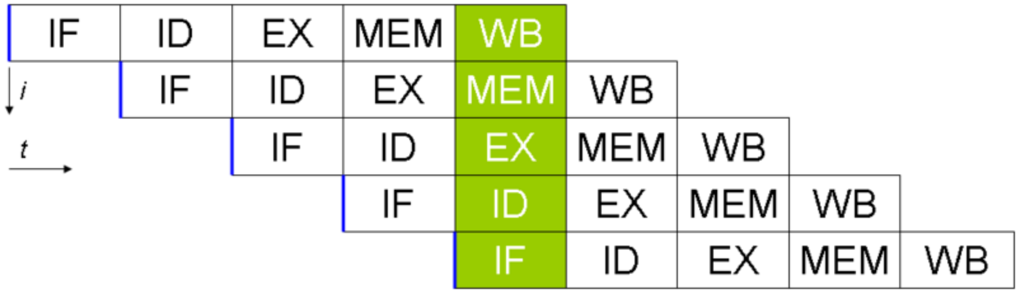
CPI for a pipelined processor:

Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

Basic block: a straight-line code with no branches.

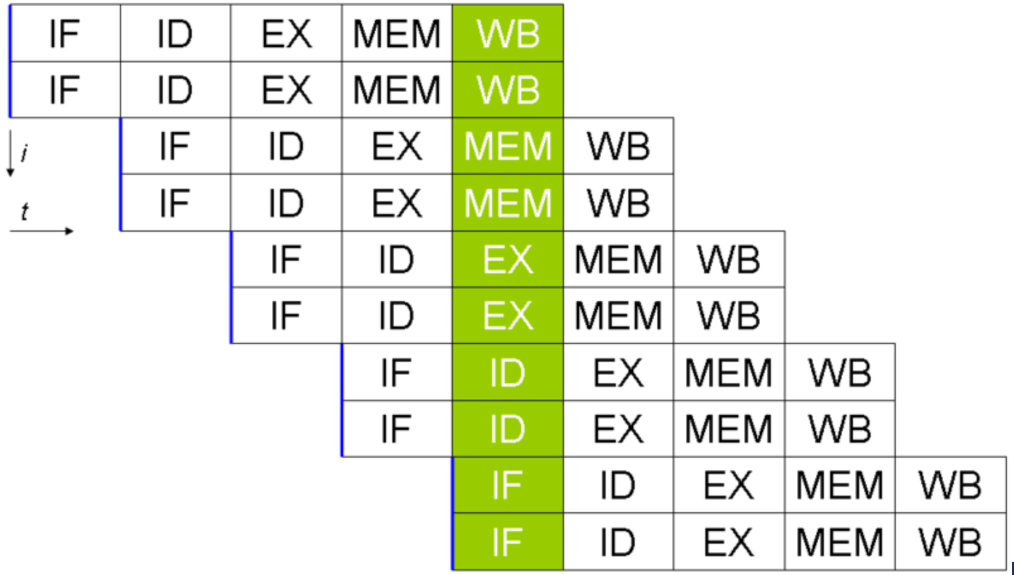
- Typical size between three to six instructions.
- Too small to exploit significant amount of parallelism.
- We must exploit ILP across multiple basic blocks.

Pipelining



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Superscalar



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Identifying parallel instructions

- Hardware Techniques
  - ✓ Out of order execution
    - Window Size
  - ✓ Speculative execution
    - Branch Prediction
    - Branch Fanout

- Compiler Techniques
  - ✓ Register Renaming

    ```
    ADD $t0,$s1,$2
    SW $t0, 0($s3)
    ADD $t0,s$4,$s5
    SW $t0, 0($s6)
    ```
  - ✓ Unrolling loops
    - Takes advantage of loop level parallelism

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Hardware speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls

Loop-level parallelism exploits parallelism among iterations of a loop. A completely parallel loop adding two 1000-element arrays:

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

Within an iteration there is no opportunity for overlap, but every iteration can overlap with any other iteration.

The loop can be unrolled either statically by compiler or dynamically by hardware.

Vector processing is also possible. Supported in DSP, graphics, and multimedia applications.

If two instructions are **parallel**, they can be executed simultaneously in a pipeline without causing any stalls, assuming the pipeline has sufficient resources.

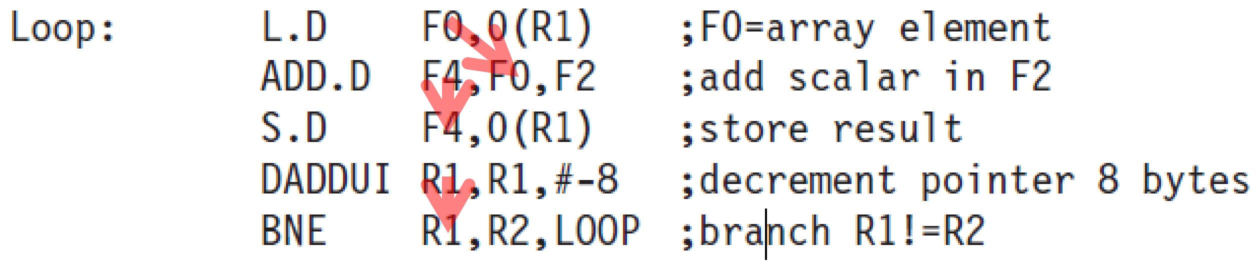
Two dependent instructions must be executed in order, but can often be partially overlapped.

Three types of dependences: **data dependences**, **name dependences**, and **control dependences**.

Instruction j is **data dependent** on instruction i if:

- i produces a result that may be used by j, or
- j is data dependent on an instruction k, and k is data dependent on i.

The following loop increments a vector of values in memory starting at 0(R1), with the last element at 8(R2)), by a scalar in register F2.



```
Loop:       L.D     F0,0(R1)      ;F0=array element
            ADD.D   F4,F0,F2      ;add scalar in F2
            S.D     F4,0(R1)      ;store result
            DADDUI  R1,R1,#-8     ;decrement pointer 8 bytes
            BNE     R1,R2,LOOP    ;branch R1!=R2
```

The data dependences in this code sequence involve both floating-point and integer data.

Since between two data dependent instructions there is a chain of one or more data hazards, they cannot be executed simultaneously or completely overlap.

Data dependence conveys:

- the possibility of a hazard,
- the order in which results must be calculated, and
- an upper bound on how much parallelism can be exploited.

Detecting dependence of registers is straightforward.

- Register names are fixed in the instructions.

Dependences that flow through memory locations are more difficult to detect.

- Two addresses may refer to the same location but look different: For example, 100(R4) and 20(R6).
- The effective address of a load or store may change from one execution of the instruction to another (so that 20(R4) and 20(R4) may be different).

Name Dependence

A **name dependence** occurs when two instructions use the same register or memory location, called **name**, but there is no flow of data between the instructions. If ***i*** precedes ***j*** in program order:

Anti dependence between ***i*** and ***j*** occurs when ***j*** writes a register or memory location that ***i*** reads.

```
S.D      F4,0(R1)     ;store result
DADDUI   R1,R1,#-8    ;decrement pointer 8 bytes
```

The original ordering must be preserved to ensure that ***i*** reads the correct value.

Output dependence occurs when ***i*** and ***j*** write the same register or memory location. Their ordering must be preserved to ensure proper value written by ***j***.

Name dependence is not a true dependence.

- The instructions involved can execute simultaneously or be reordered.
- The name (register # or memory location) is changed so the instructions do not conflict.

Register renaming can be more easily done.

- Done either statically by a compiler or dynamically by the hardware.

Data Hazards

A hazard is created whenever a dependence between instructions is close enough.

- **Program order** must be preserved.

The goal of both SW and HW techniques is to exploit parallelism by preserving program order **only where it affects the outcome of the program**.

- Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards are classified depending on the order of read and write accesses in the instructions. Consider two instructions ***i*** and ***j***, with ***i*** preceding ***j***

The possible data hazards are: vinay

RAW (**read after write**). ***j*** tries to read a source before ***i*** writes it.

- The most common, corresponding to a true data dependence.
- Program order must be preserved.

WAW (**write after write**). ***j*** tries to write an operand before it is written by ***i***.

- Writes are performed in the wrong order, leaving the value written by ***i*** rather than by ***j***.
- Corresponds to an output dependence.
- Present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

WAR (**write after read**). ***j*** tries to write a destination before it is read by ***i***, so ***i*** incorrectly gets the new value.

- Arises from **anti dependence**.
- Cannot occur in most static issue pipelines because all reads are early (in ID) and all writes are late (in WB).
- Occurs when there are some instructions that write results early in the pipeline and other instructions that read a source late in the pipeline.
- Occurs also when instructions are reordered.

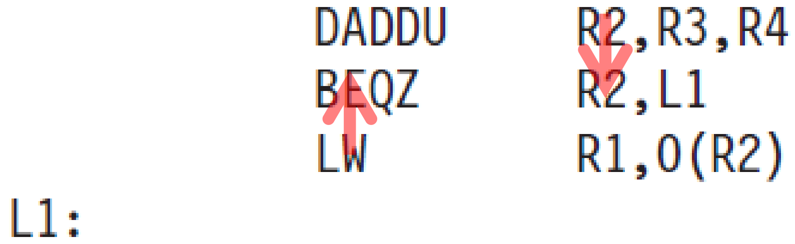
Control Hazards

A **control dependence** determines the ordering of ***i*** with respect to a branch so that ***i*** is executed in correct order and only when it should be.

There are two constraints imposed by control dependences:

- An instruction that is control dependent on a branch cannot be moved **before** the branch so that its execution **is no longer controlled** by the branch.
- An instruction that is **not** control dependent on a branch cannot be moved **after** the branch so that its execution **is controlled** by the branch.

Consider this code:



```
        DADDU     R2,R3,R4
        BEQZ      R2,L1
        LW        R1,0(R2)
L1:
```

If we do not maintain the **data dependence** involving R2, the result of the program can be changed.

If we ignore the **control dependence** and move the load before the branch, the load may cause a memory protection exception. (why?)

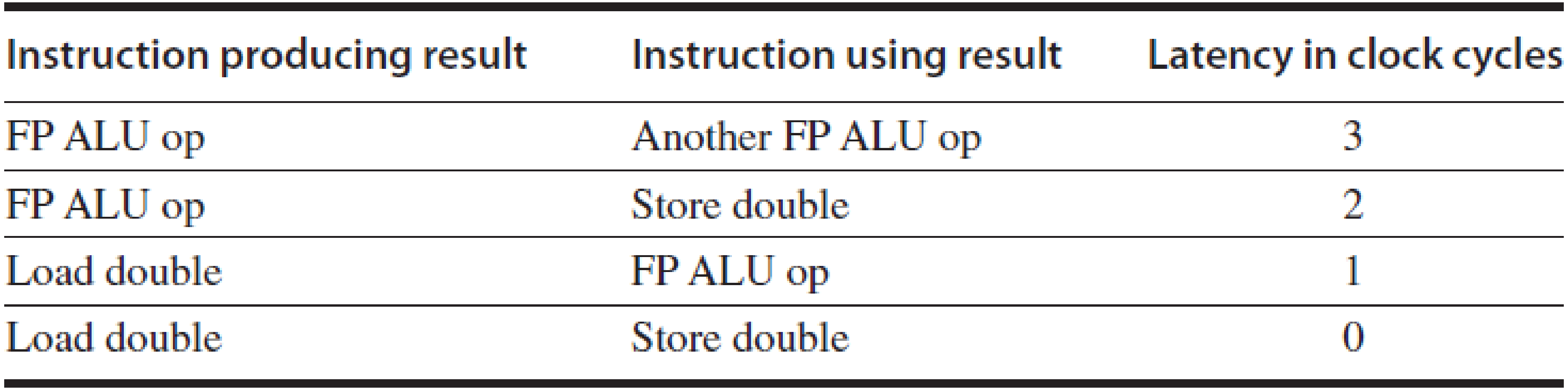
It is not **data dependence** preventing interchanging the BEQZ and the LW; it is only the **control dependence**.

Compiler Techniques for Exposing ILP

Pipeline is kept full by finding sequences of unrelated instructions that can be overlapped in the pipeline.

To avoid stall, a dependent instruction must be separated from the source by a distance in clock cycles equal to the pipeline latency of that source.

Example: Latencies of FP operations



Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Flynn’s Classification

Flynn’s Classification

- Studied and proposed by Michael Flynn in 1972
- Did not consider the machine architecture for classification of parallel computers
- Introduced the concept of instruction and data streams for categorizing of computers
- All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn’s classification

Instruction Cycle

- Consists of a sequence of steps needed for the execution of an instruction in a program.
- A typical instruction in a program is composed of two parts
  - Opcode
  - Operand
    - The Operand part specifies the data on which the specified operation is to be done
      - addressing mode
      - the Operand.

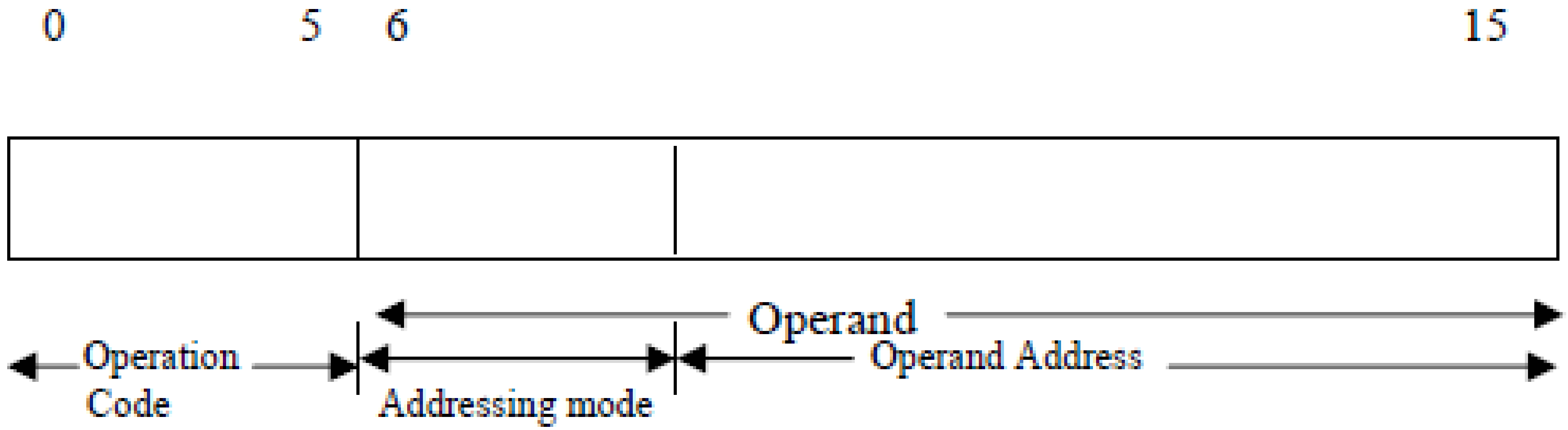


Figure: Opcode and Operand

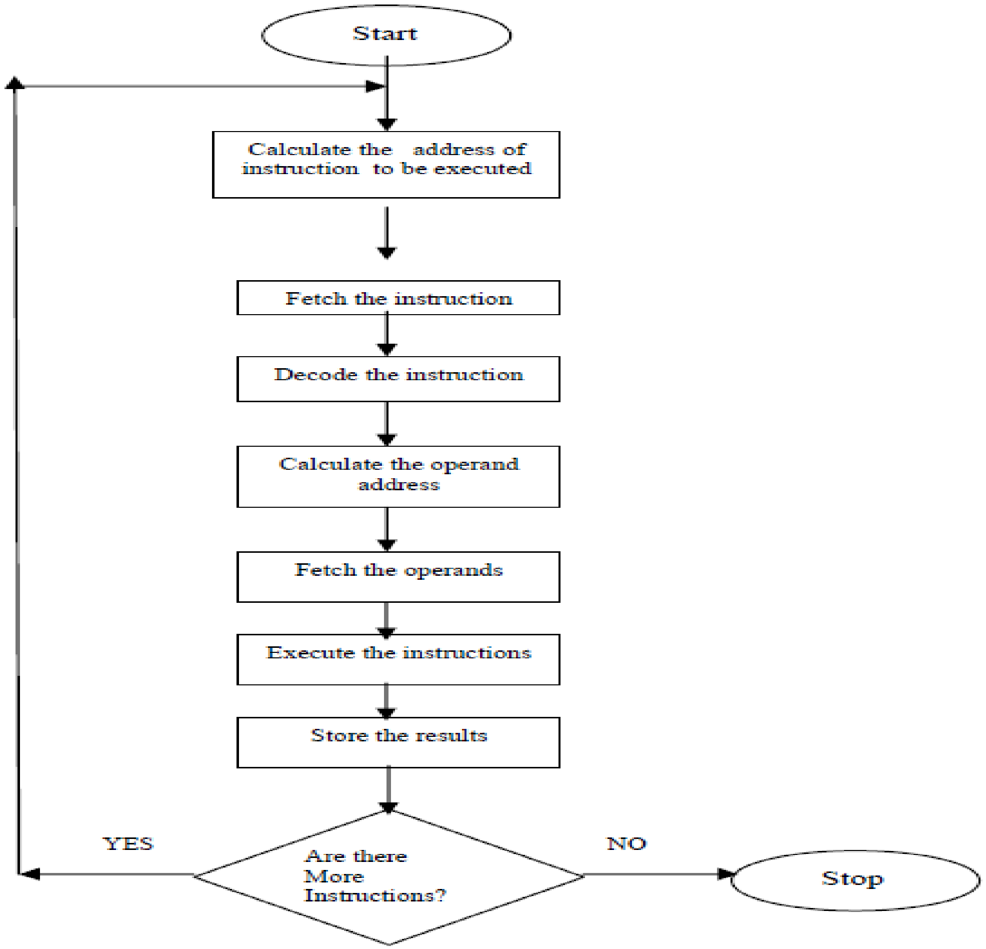


Figure: Instruction execution steps

Instruction Stream and Data Stream

- The term **‘stream’** refers to a sequence or flow of either instructions or data operated on by the computer

Instruction stream

- In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established.

Data stream

- there is a flow of operands between processor and memory bi-directionally.

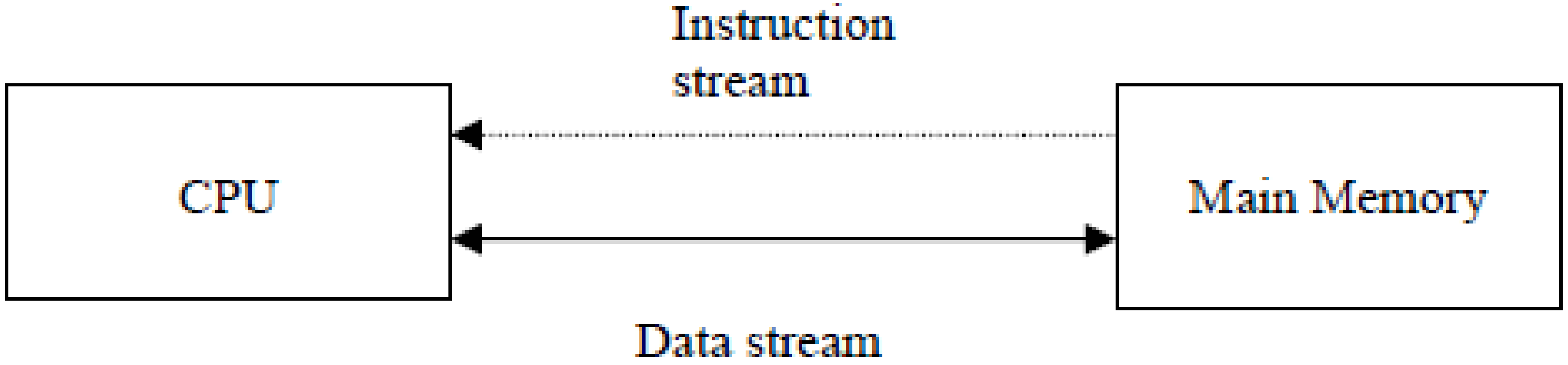


Figure: Instruction and data stream

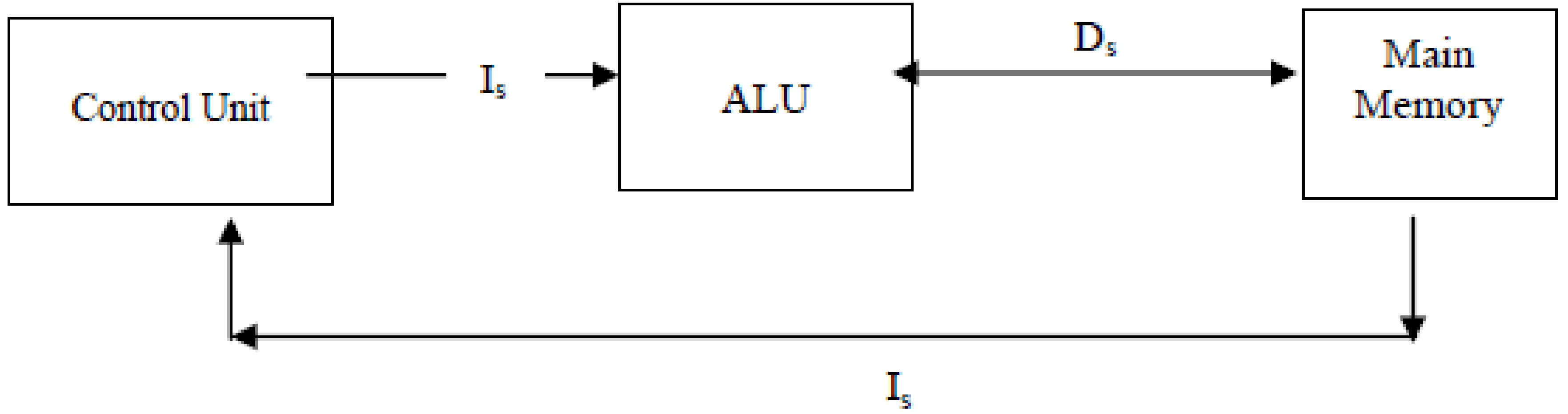
Flynn’s Classification

- Single Instruction and Single Data stream (SISD)
- Single Instruction and Multiple Data stream (SIMD)
- Multiple Instruction and Single Data stream (MISD)
- Multiple Instruction and Multiple Data stream (MIMD)

Single Instruction and Single Data stream (SISD)

- Sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit
- SISD machines are conventional serial computers that process only one stream of instructions and one stream of data

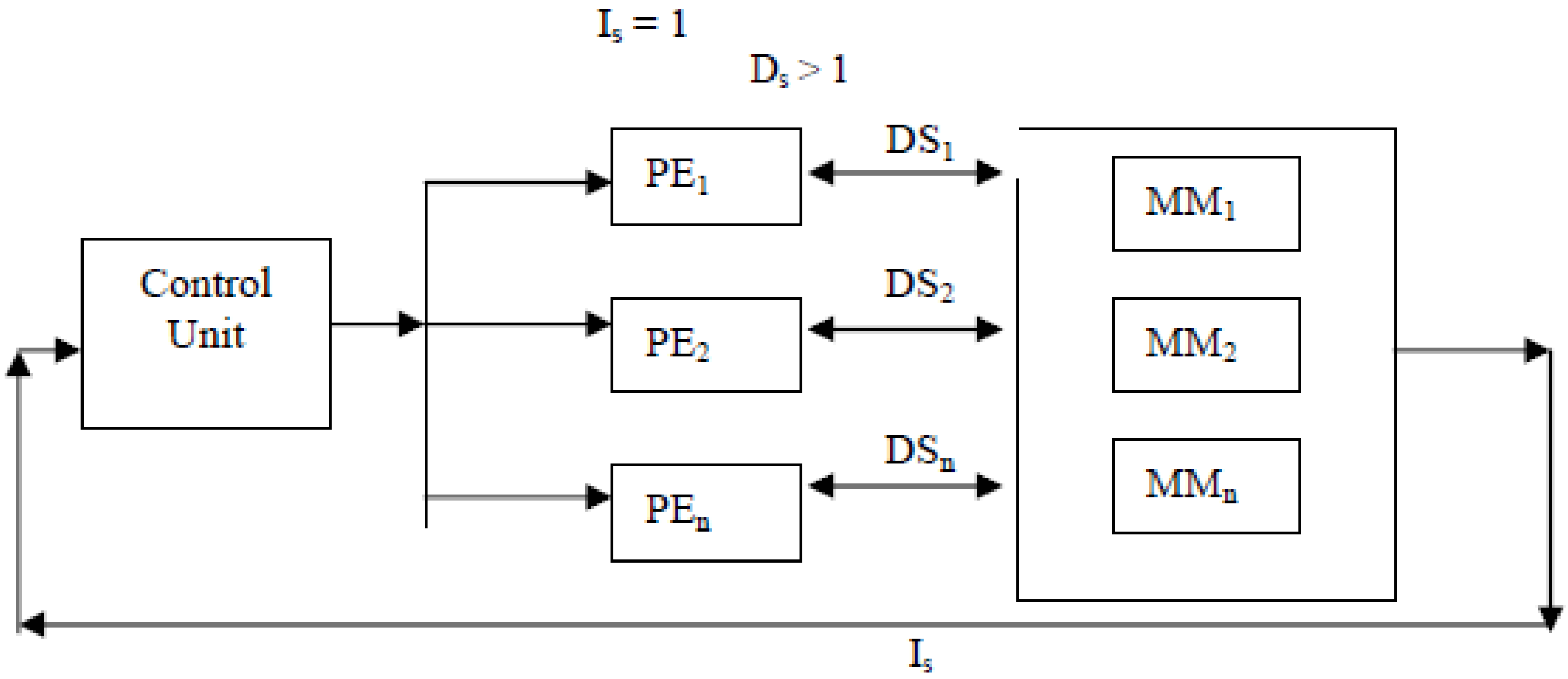
$I_s = D_s = 1$



Single Instruction and Multiple Data stream (SIMD)

- Multiple processing elements work under the control of a single control unit
- One instruction and multiple data stream
- All the processing elements of this organization receive the same instruction broadcast from the CU
- Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory
- Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together
- Each processor takes the data from its own memory and hence it has on distinct data streams.
- Every processor must be allowed to complete its instruction before the next instruction is taken for execution.
- Thus, the execution of instructions is synchronous

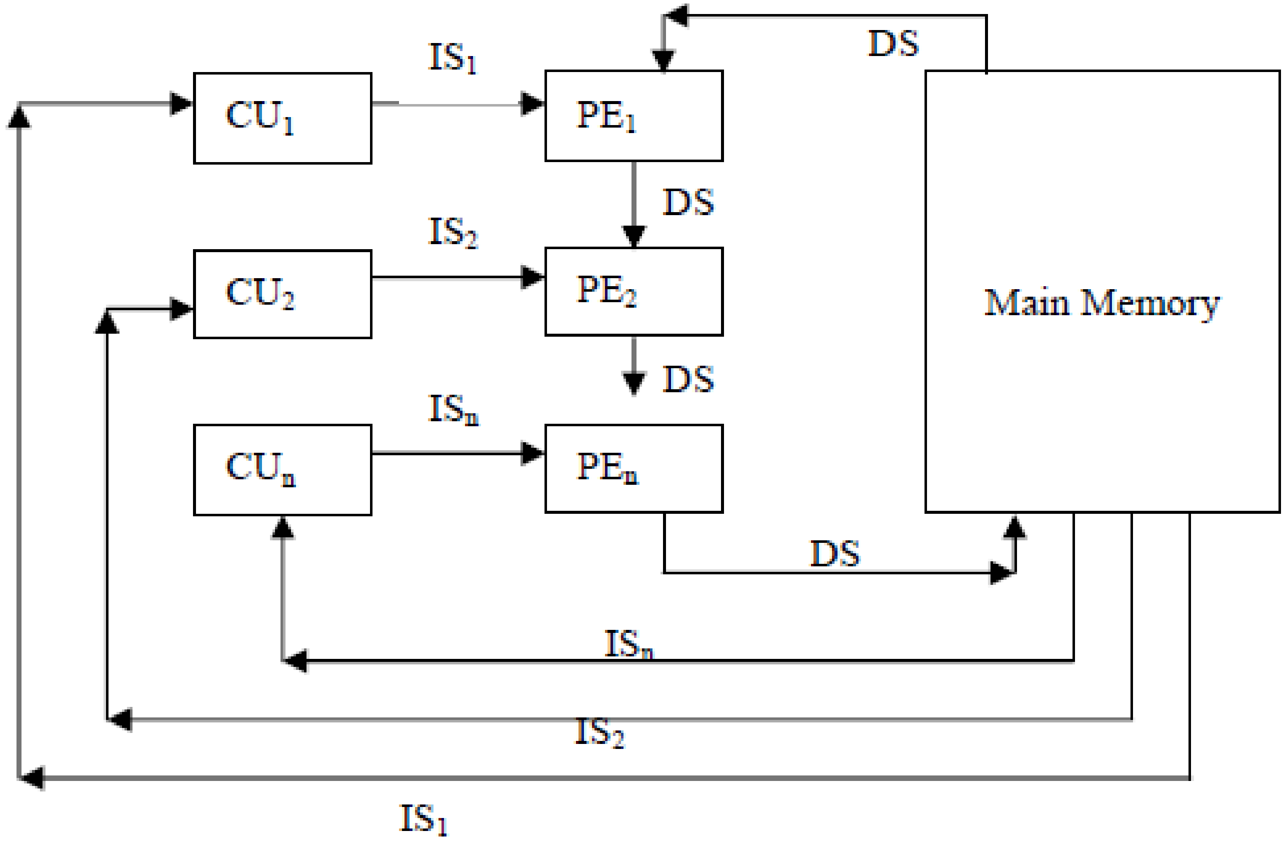
Single Instruction and Multiple Data stream (SIMD)



Multiple Instruction and Single Data stream (MISD)

- multiple processing elements are organised under the control of multiple control units
- Each control unit is handling one instruction stream and processed through its corresponding processing element
- But each processing element is processing only a single data stream at a time.
- Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification.
- All processing elements are interacting with the common shared memory for the organisation of single data stream
- This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied

Multiple Instruction and Single Data stream (MISD)



Multiple Instruction and Single Data stream (MISD)

- This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied
- But for the specialized applications, MISD organisation can be very helpful.

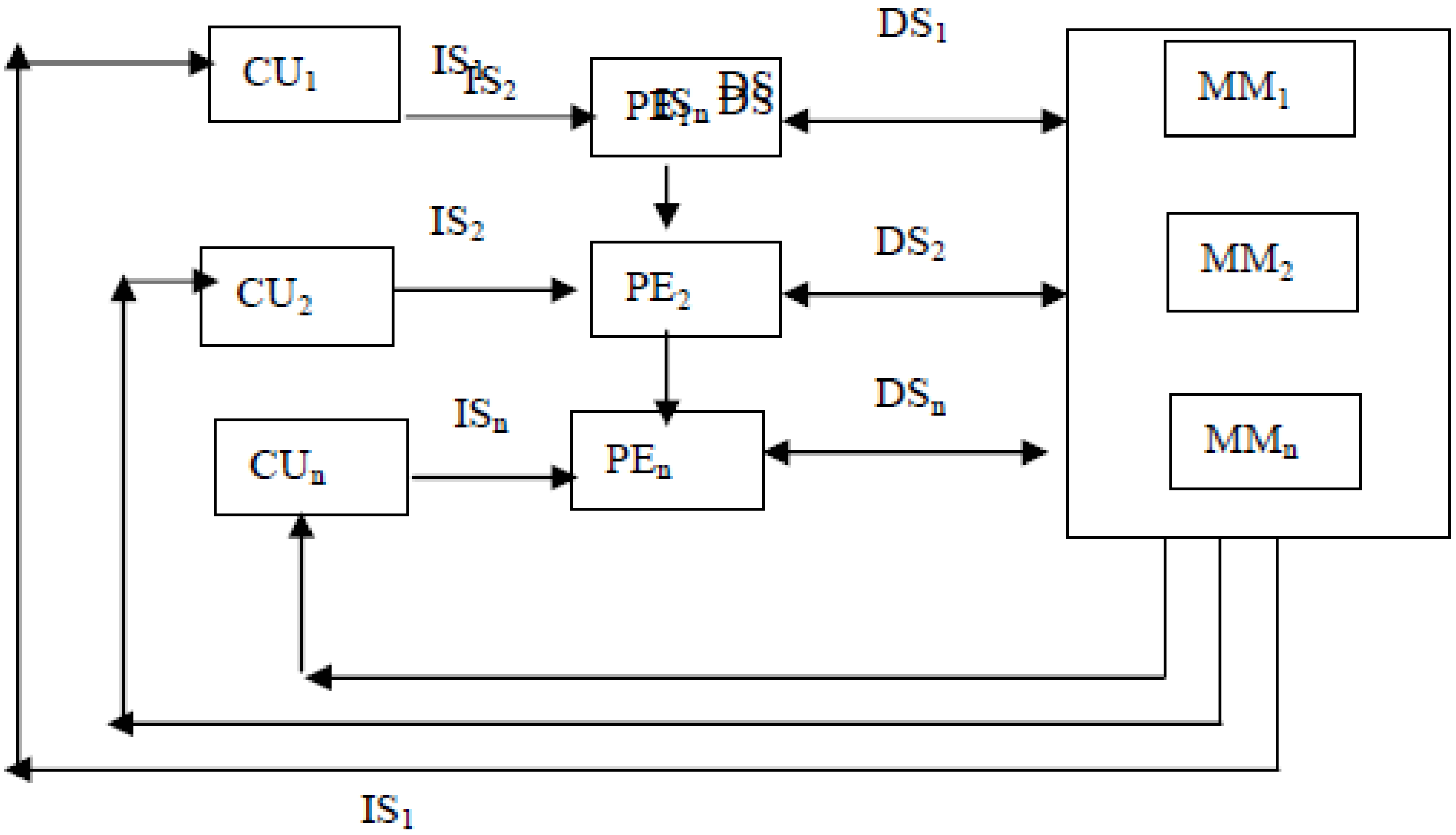
Eg.

- Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data.
- This is also known as N- version programming.
- All these redundant data are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

Multiple Instruction and Multiple Data stream (MIMD)

- Multiple processing elements and multiple control units are organized as in MISD
- Difference is that now in this organization multiple instruction streams operate on multiple data streams
- Handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory
- The processors work on their own data with their own instructions
- Tasks executed by different processors can start or finish at different times
- Not lock-stepped, as in SIMD computers, but run asynchronously
- This classification actually recognizes the parallel computer. That means in the real sense MIMD organisation is said to be a Parallel computer.
- All multiprocessor systems fall under this classification.

Multiple Instruction and Multiple Data stream (MIMD)



Hardware Multithreading

Hardware multithreading

- **Hardware multithreading** allows multiple threads to share the functional units of a single processor in an overlapping fashion.
- To permit this sharing, the processor must duplicate the independent state of each thread.
- There are two main approaches to hardware multithreading.
    - **Fine-grained multithreading**
    - **Coarse-grained multithreading**

Fine-grained multithreading

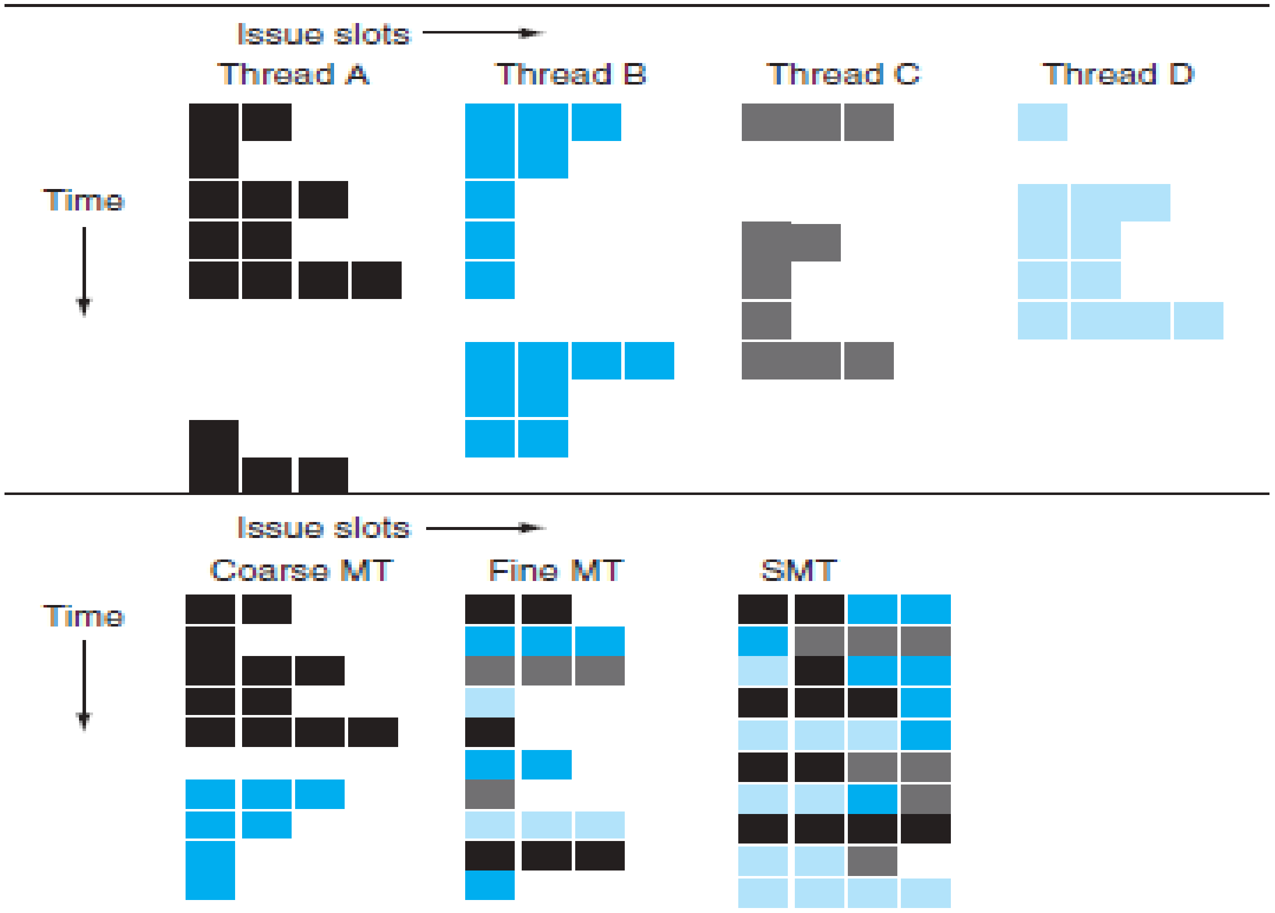
- switches between threads on each instruction, resulting in interleaved execution of multiple threads.
- interleaving is often done in a roundrobin fashion, skipping any threads that are stalled at that time
- To make fine-grained multithreading practical, the processor must be able to switch threads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

Coarse-grained multithreading

- An alternative to fine-grained Multithreading
- Switches threads only on costly stalls, such as second-level cache misses.
- This change relieves the need to have thread switching be essentially free and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall.
- Disadv.
  - it is limited in its ability to overcome throughput losses, especially from shorter stalls

Simultaneous multithreading

- Variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism.
- The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use.
- Since you are relying on the existing dynamic mechanisms, SMT does not switch resources every cycle


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

Muti-core Processors

Introduction

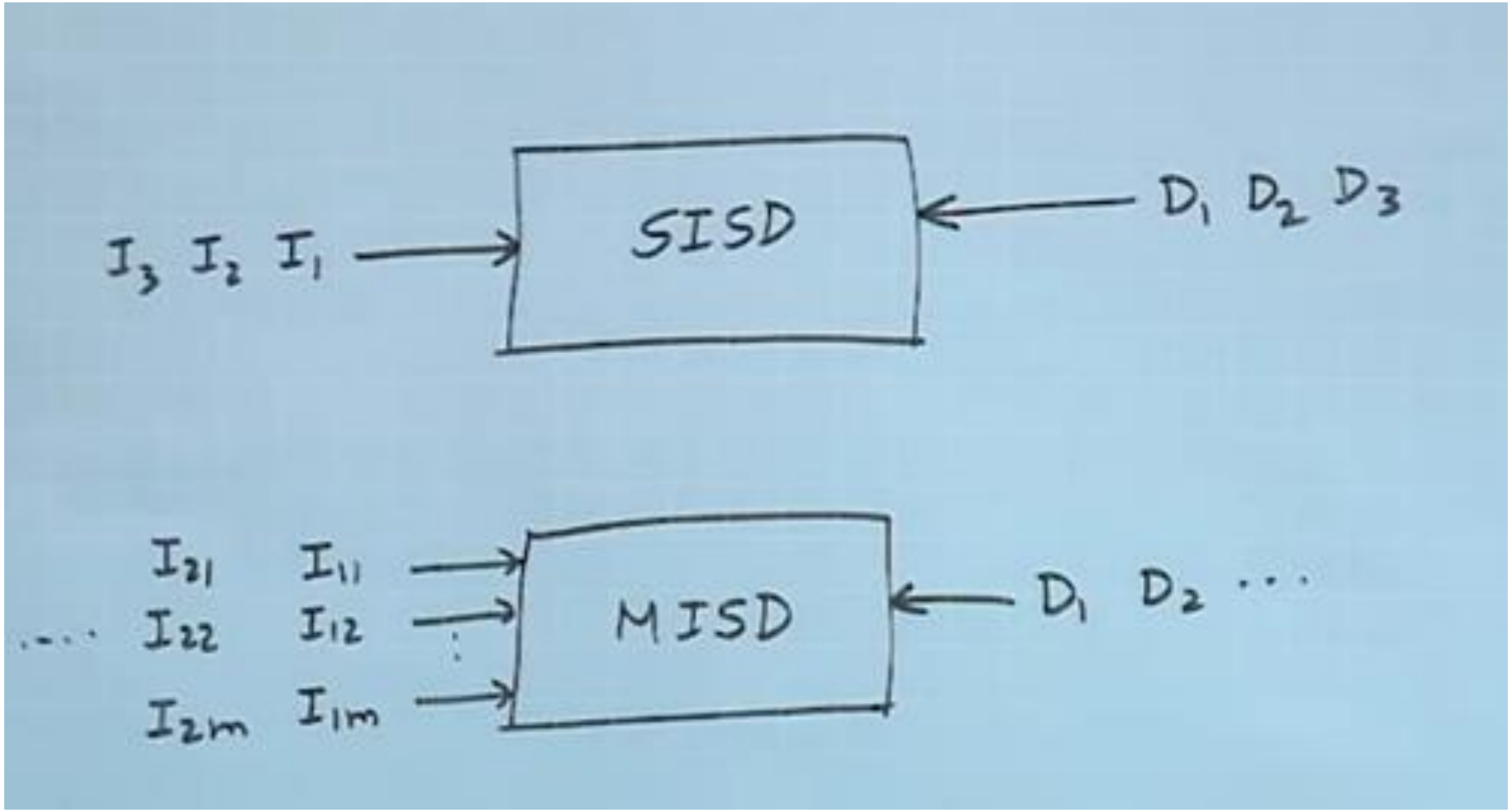
- Multi-core Processor
  - A processing system composed of two or more independent cores or CPUs
  - The cores are typically integrated onto a single integrated circuit die, or they may be integrated on multiple dies in a single-chip package.
- Cores share memory:
  - In modern multi-core systems, typically L1 and L2 cache are private to each core, while the L3 cache is shared among the cores
- In symmetric multi-core systems, all the cores are identical
  - Example: multi-core processors used in computer systems
- In asymmetry multi-core systems, the cores may have different functionalities

Why Multi-core?

- It is difficult to sustain Moore’s law and at the same time meet performance demands of various applications.
    - Difficult to increase clock frequency, mainly due to power consumption issues.
- Possible solutions:
    - Replicate hardware and run them at a lower clock rate to reduce power consumption.
    - 1 core running at 3 GHz has the same performance as 2 cores running at 1.5 GHz, with lower power consumption.

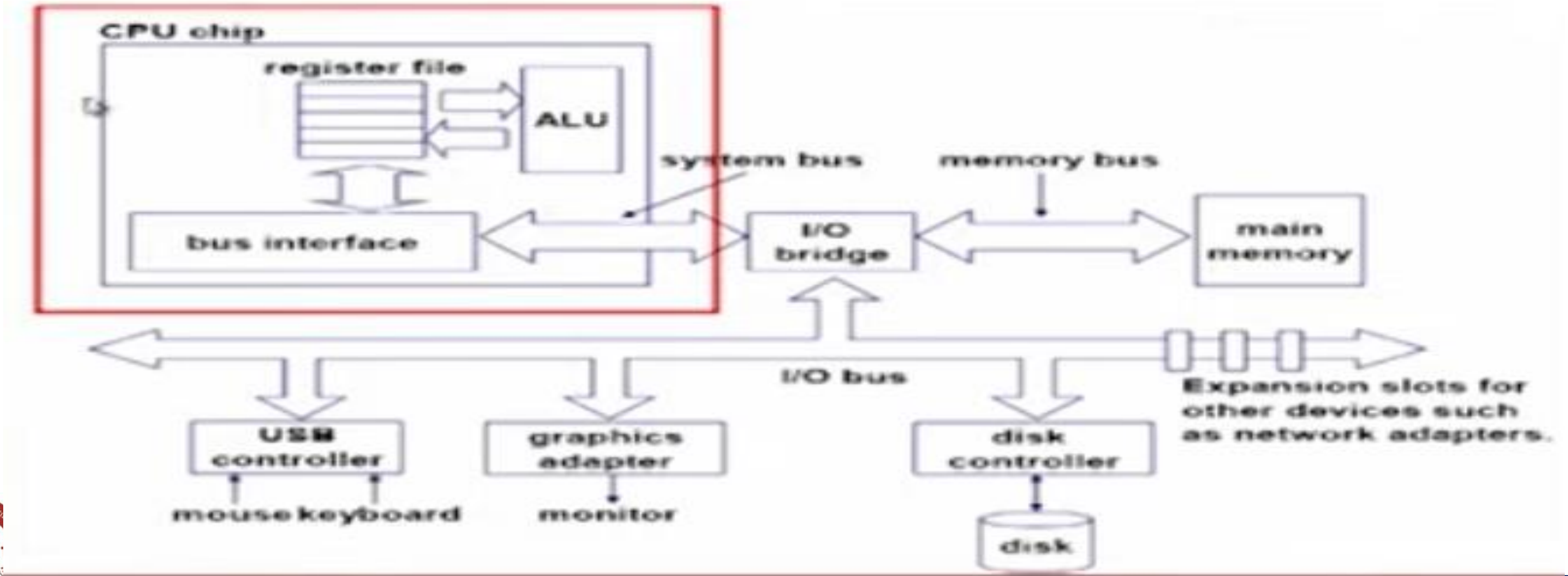
Taxonomy of Parallel Architectures

- Single Instruction and Single Data stream (SISD)
  - Traditional uniprocessor systems
- Single Instruction and Multiple Data stream (SIMD)
  - Array and Vector processors
- Multiple Instruction and Single Data stream (MISD)
  - No commercial implementation exists
  - Pipelining can be argued as a type of MISD processing
- Multiple Instruction and Multiple Data stream (MIMD)
  - Multiprocessor systems (various architectures exist).

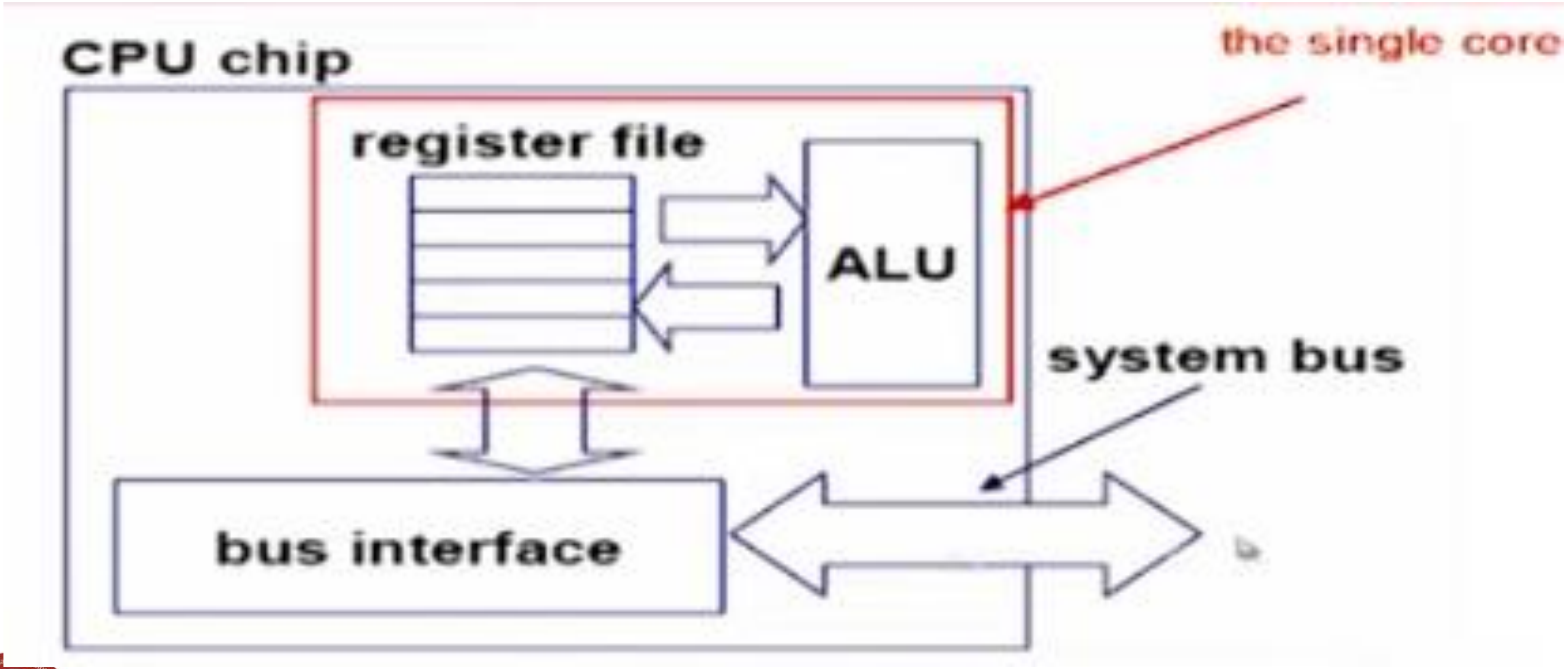

SISD
I_3 I_2 I_1
D_1 D_2 D_3
MISD
I_{21} I_{11}
.... I_{22} I_{12}
I_{2m} I_{1m}
D_1 D_2 ...

Single-core Computers

- Falls under SISD category
- Typically two buses
  - A high-speed CPU-memory bus, that also connects to I/O bridge.
  - A lower-speed I/O bus, connecting various peripherals.

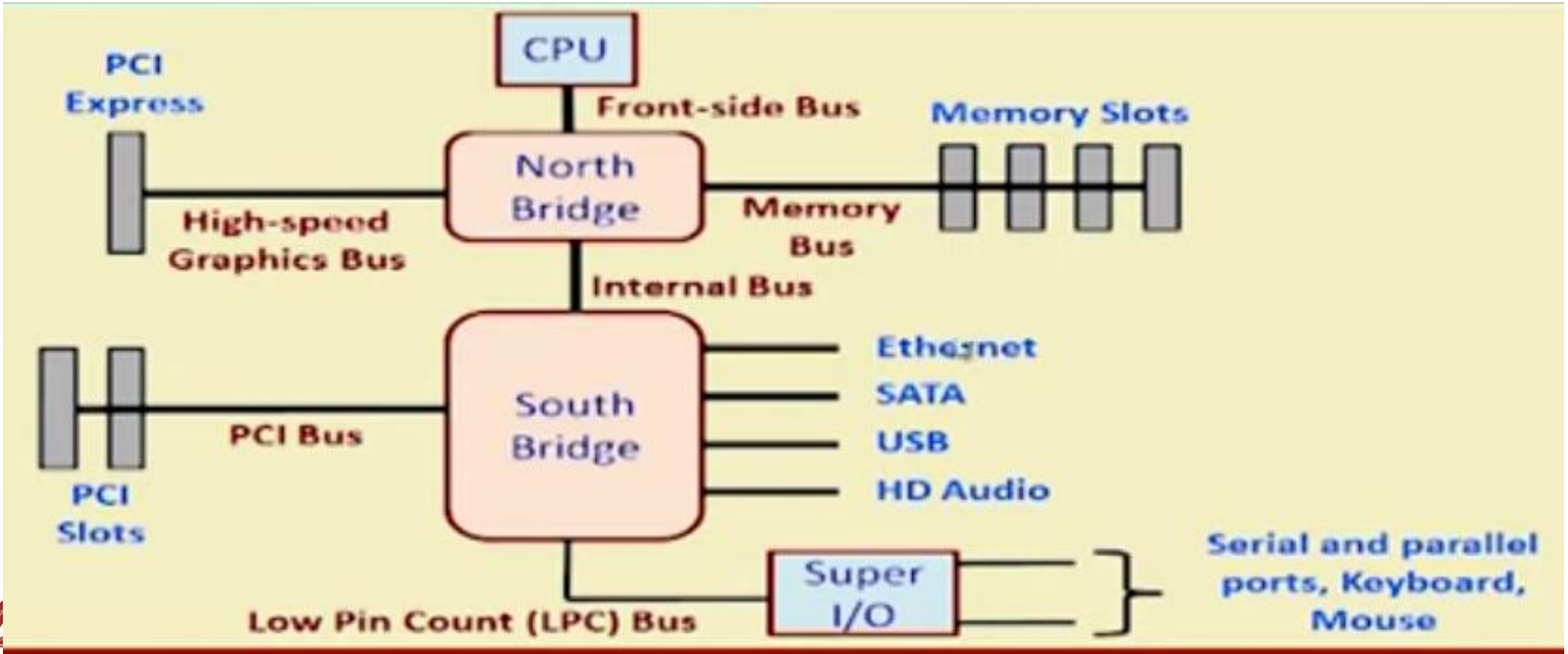


Single-core processor



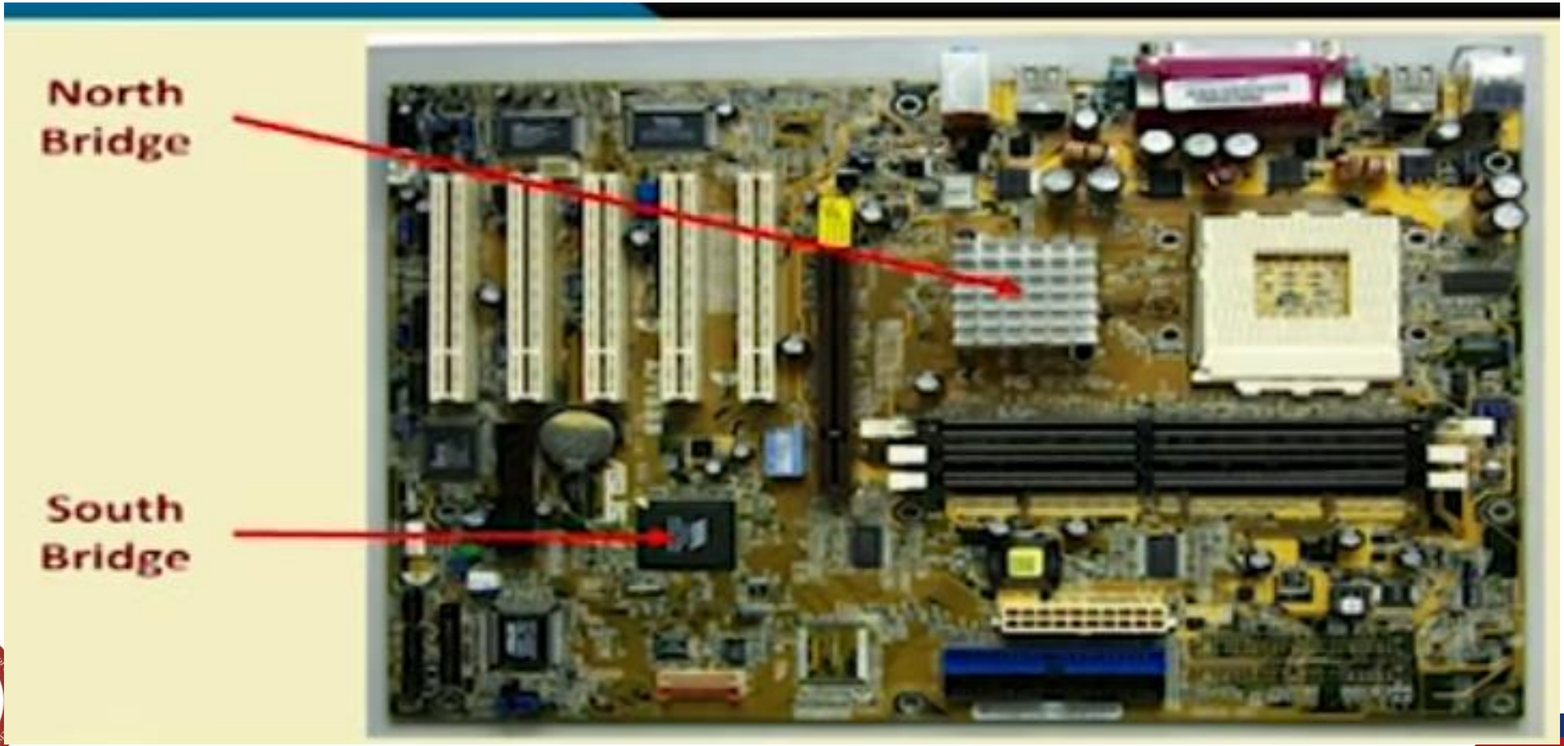
Typical mother board architecture

- Chipset consisting of north bridge and south bridge

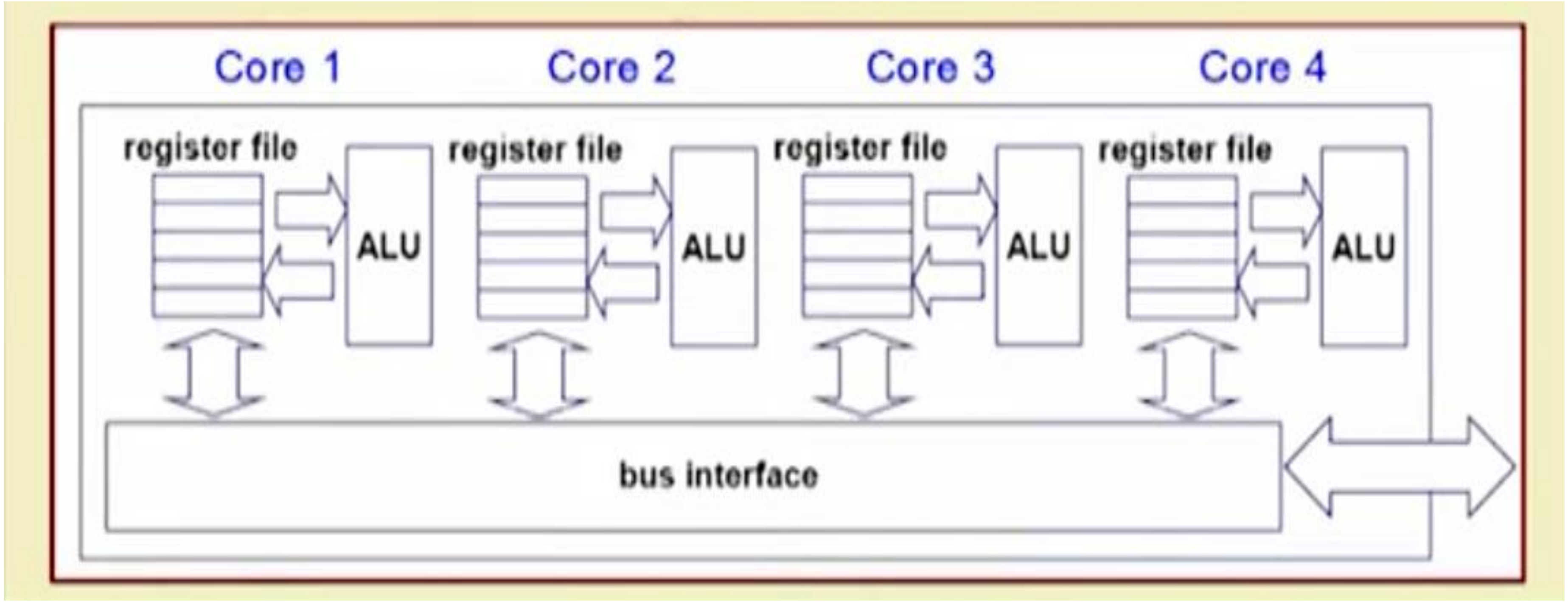


Locating north bridge and south bridge chipset on Motherboard

- Bus speed and other capabilities depend upon the chipset.



Multi-core Architecture

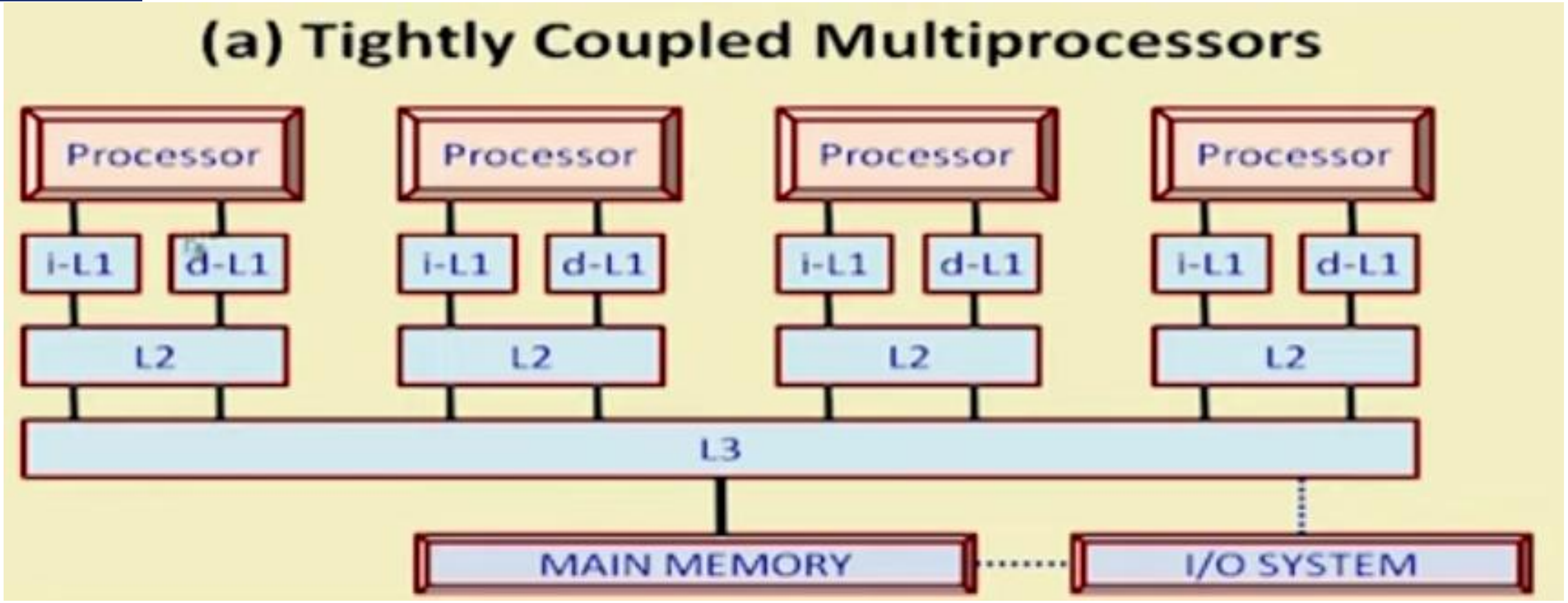


Traditional Multiprocessor Architecture

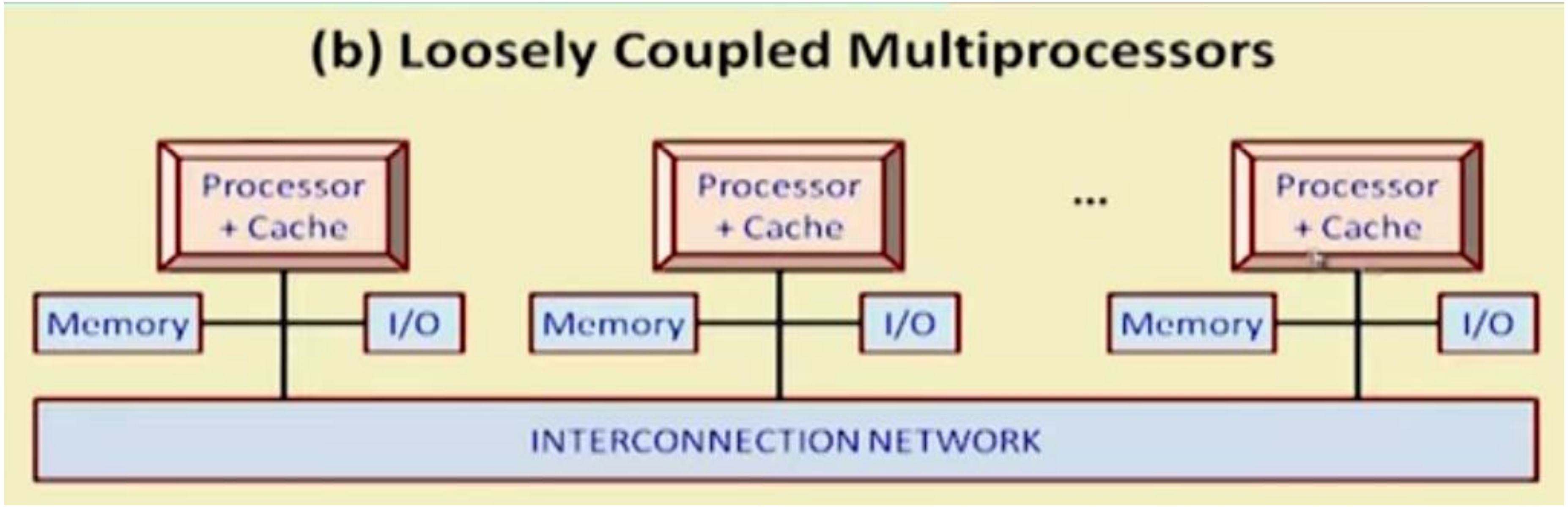
Can be broadly classified into two types:

a. Tightly coupled multiprocessors
   - The processors access common shared memory
   - Inter-processor communication takes place through shared memory
   - Multi-core architectures fall under this category

b. Loosely coupled multiprocessors
   - Memory is distributed among the processors.
   - Processors typically communicate through a high-speed interconnection network



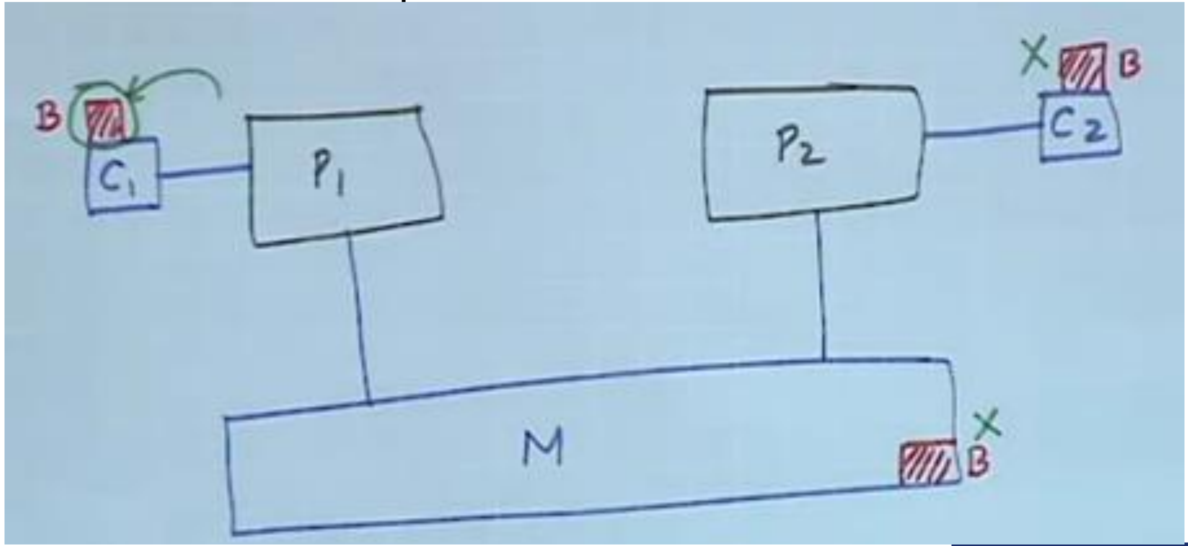
- Some features
  - Difficult to extend it to large number of processors
  - Memory bandwidth requirements increase with the number of processors
  - Memory access time for all processors is uniform.
    - Called *Uniform Memory Access – UMA*



- Some features
  - Cost-effective way to scale memory bandwidth.
  - Communicating data between processors is complex and has higher latency.
  - Memory access time depends on the location of data.
    - Called *Non-Uniform Memory Access – NUMA*

Cache Coherence Problem in Multiprocessors

- Maintaining coherence between data loaded in processor caches is an issue in multiprocessor system.
  - Same memory block is loaded into two processor caches.
  - One of the processors updates the data in its local caches.
  - Data in the other processor cache and also memory becomes inconsistent
- Broadly two classes techniques are used to solve this problem:
  - Snoopy protocols
  - Directory-based protocols


B
C1
P1
P2
X
B
C2
M
X
B