GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING (Autonomous)
Approved by AICTE, New Delhi and Affiliated to JNTU-Kakinada
Re-accredited by NAAC with "A" Grade with a CGPA of 3.47/4.00
Madhurawada, Visakhapatnam - 530 048.

COMPUTER ORGANIZATION
(19EC11D4)

Mrs.N.Santoshi
Assistant Professor
ECE Department

UNIT IV

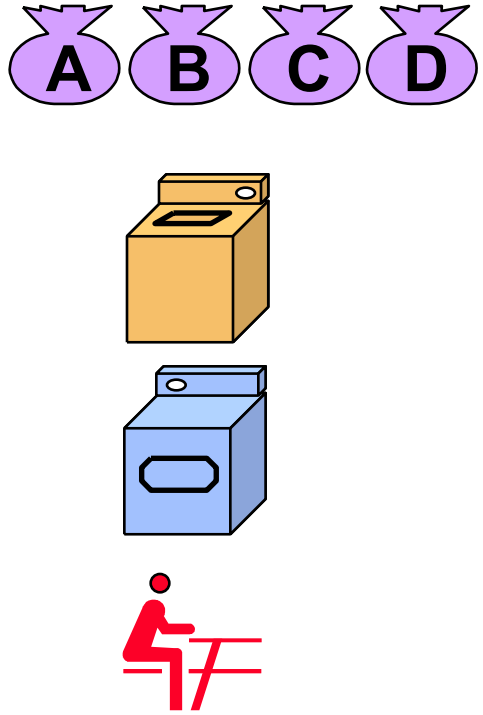
PIPELINING AND PARALLEL PROCESSING

Making the Execution of Programs Faster

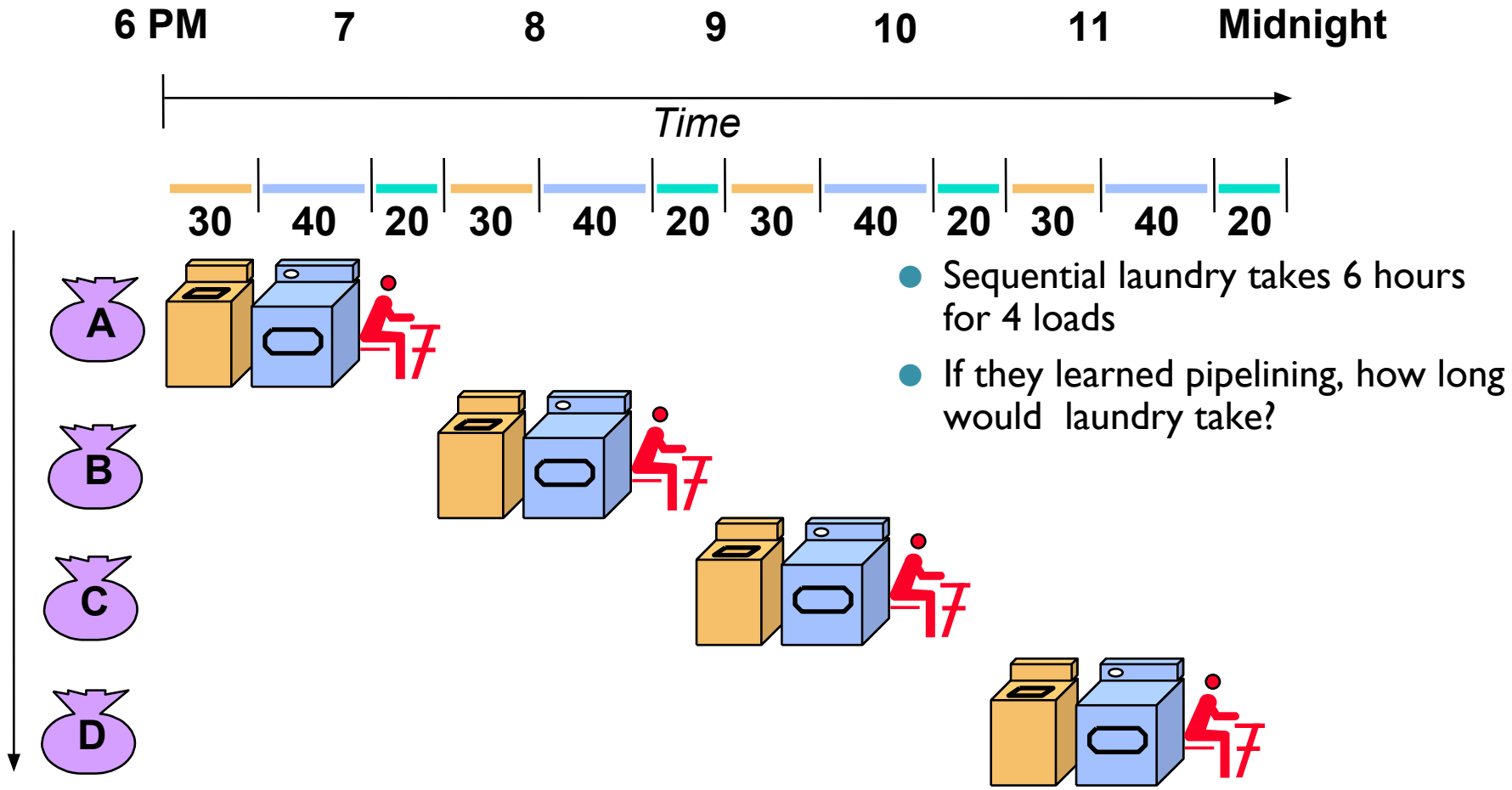
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

Traditional Pipeline Concept

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes

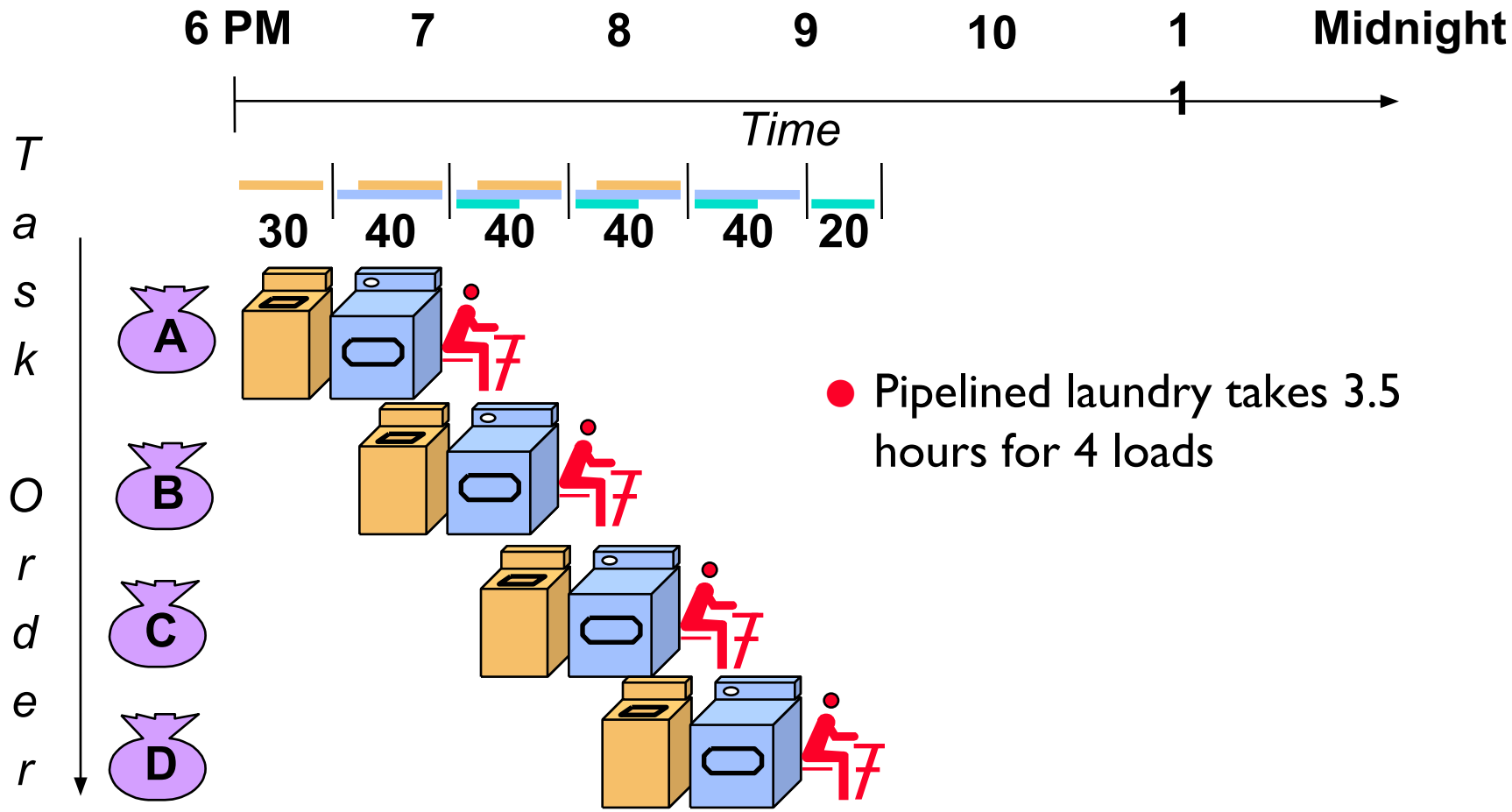


Traditional Pipeline Concept



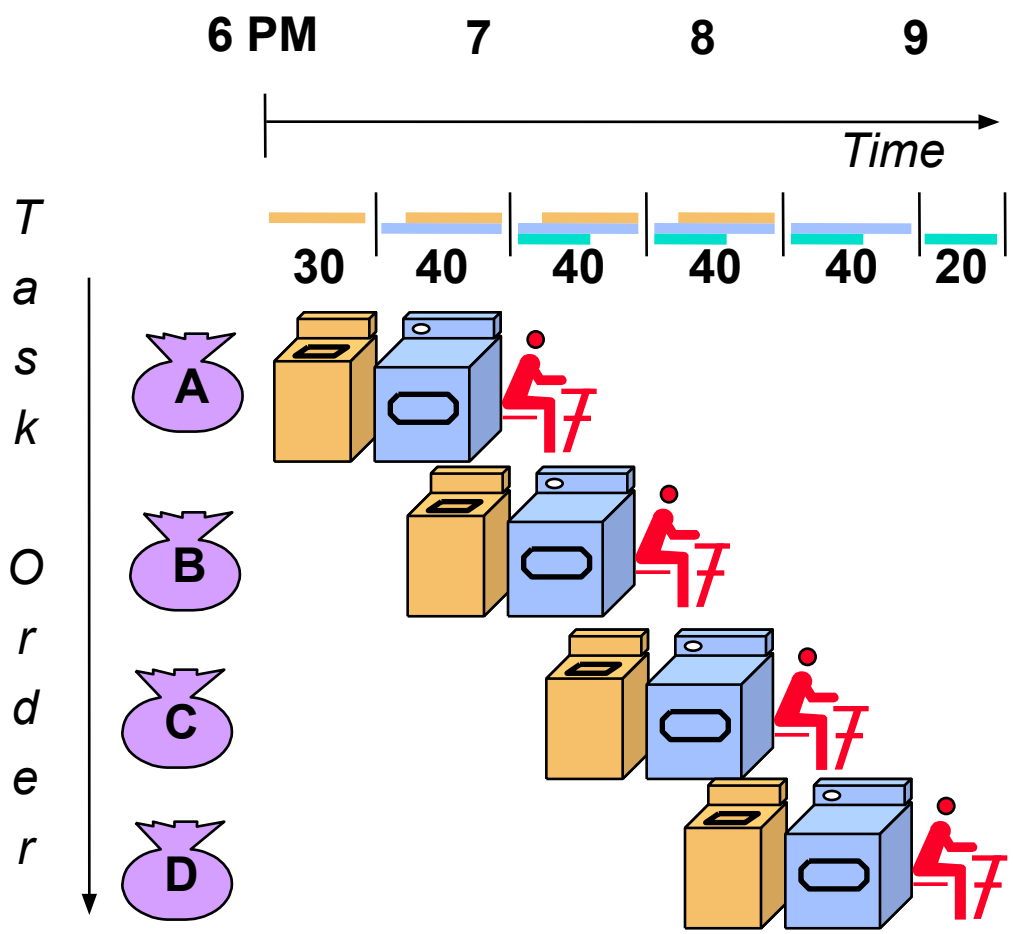
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



- Pipelined laundry takes 3.5 hours for 4 loads

Traditional Pipeline Concept

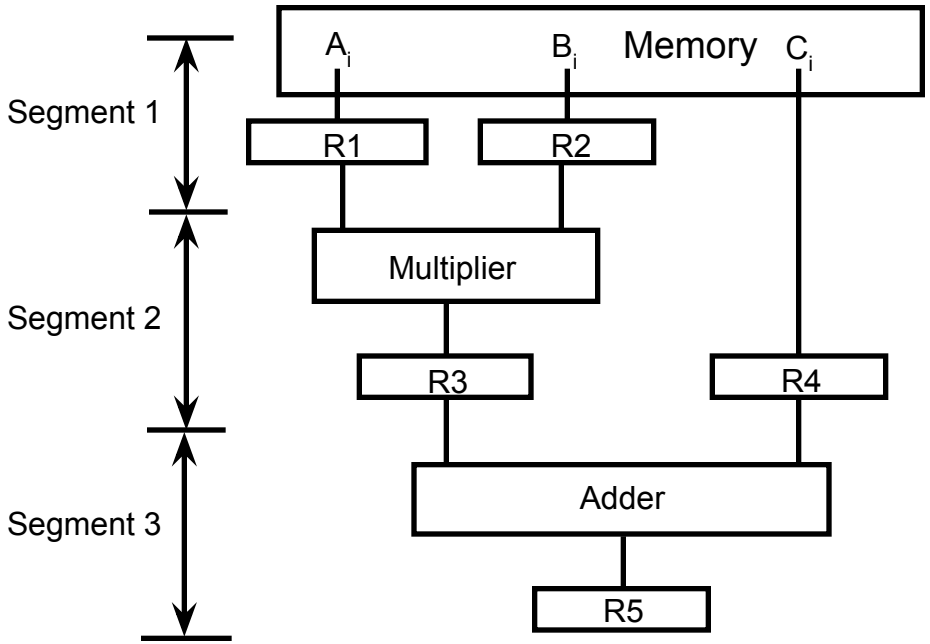


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7



$R1 \leftarrow A_i$, $R2 \leftarrow B_i$ Load A_i and B_i

$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$ Multiply and load C_i

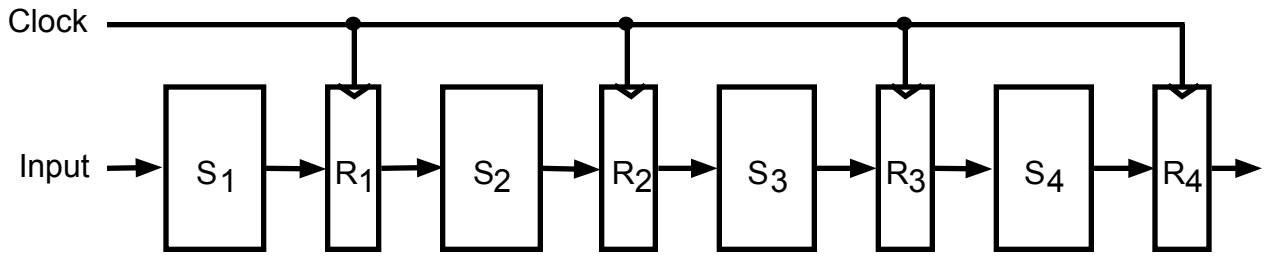
$R5 \leftarrow R3 + R4$ Add

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

Segment	1	2	3	4	5	6	7	8	9
1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

Clock cycles

PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

τ_1: Time required to complete the n tasks

$\tau_1 = n * t_n$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

τ_k: Time required to complete the n tasks

$\tau_k = (k + n - 1) * t_p$

Speedup

S_k: Speedup

$S_k = n*t_n / (k + n - 1)*t_p$

$$\lim_{n \to \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k * t_p)$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline
- subopertion in each stage; t_p = 20nS
- 100 tasks to be executed
- 1 task in non-pipelined system; 20*4 = 80nS

Pipelined System

$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$

Non-Pipelined System

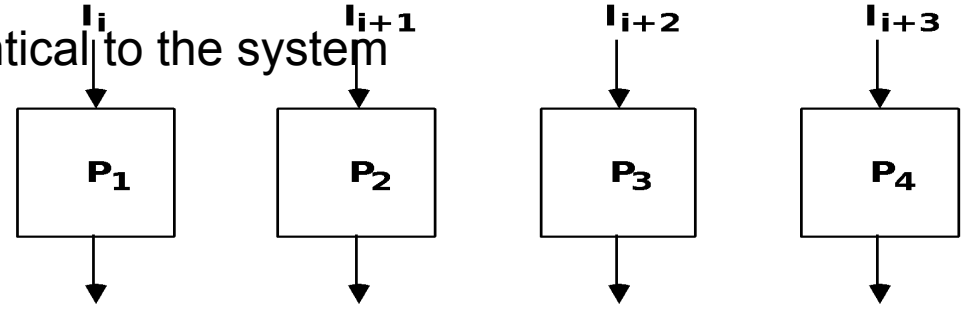
$n*k*t_p = 100 * 80 = 8000nS$

Speedup

$S_k = 8000 / 2060 = 3.88$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Multiple Functional Units

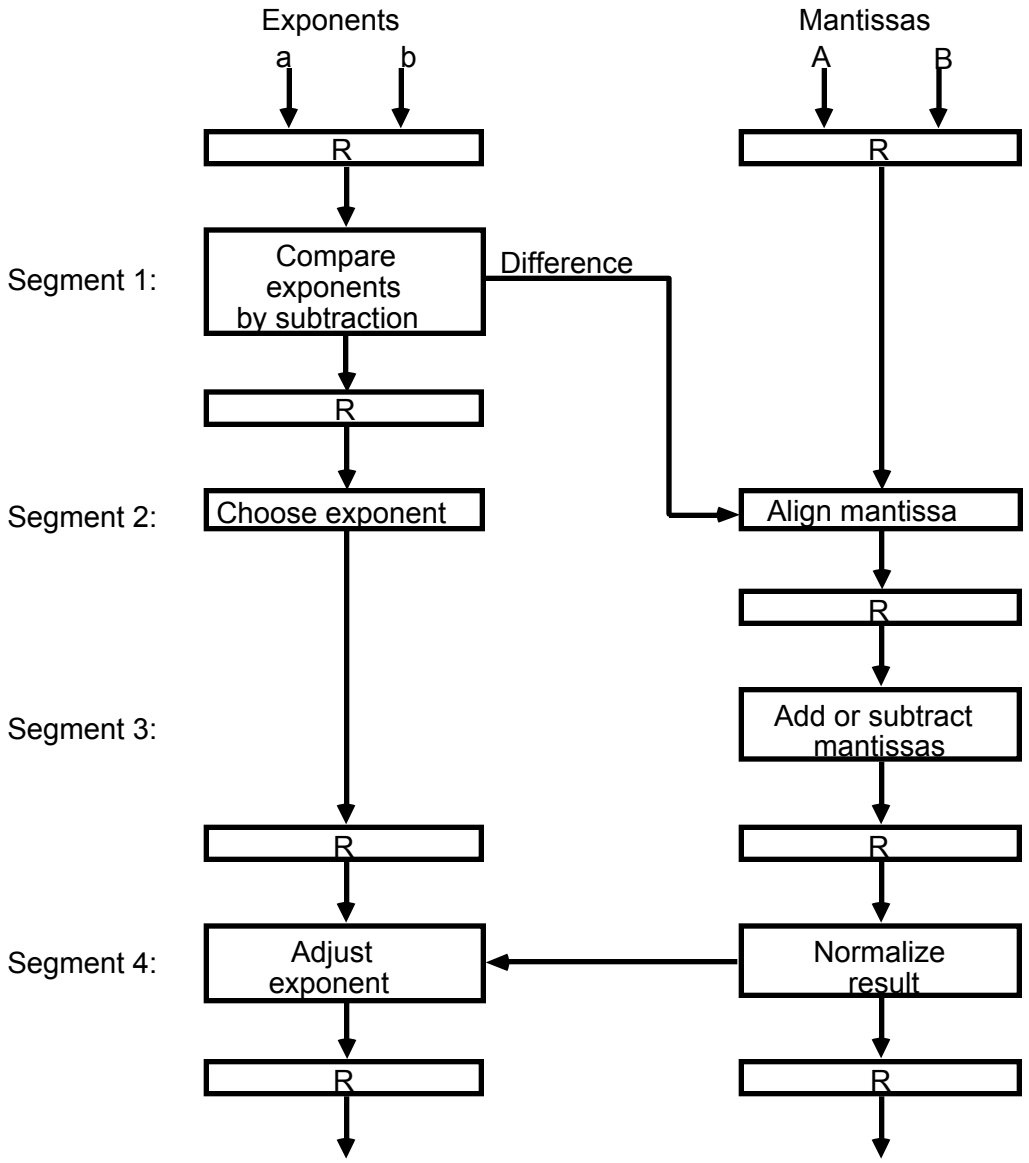


ARITHMETIC PIPELINE

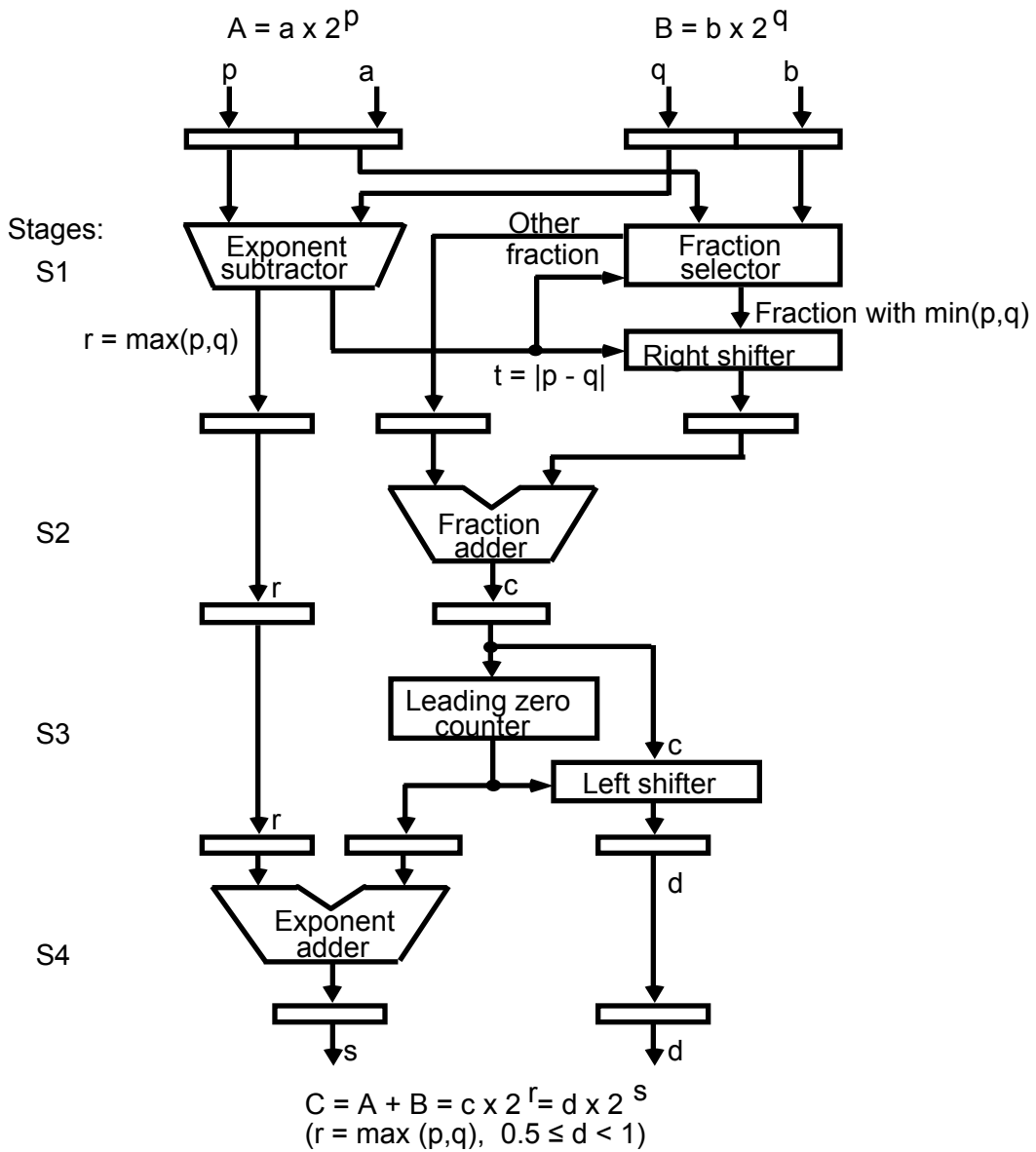
Floating-point adder

$X = A \times 2^a$
$Y = B \times 2^b$

[1] Compare the exponents
[2] Align the mantissa
[3] Add/sub the mantissa
[4] Normalize the result



4-STAGE FLOATING POINT ADDER



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

* Some instructions skip some phases
* Effective address calculation can be done in the part of the decoding phase
* Storage of the operation result into a register is done automatically in the execution phase

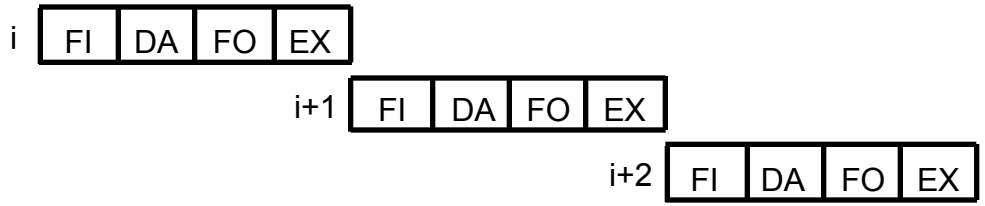
==> 4-Stage Pipeline

[1] FI: Fetch an instruction from memory
[2] DA: Decode the instruction and calculate the effective address of the operand
[3] FO: Fetch the operand
[4] EX: Execute the operation

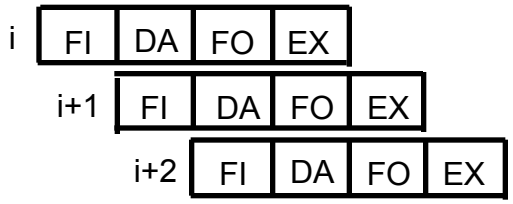
INSTRUCTION PIPELINE

Execution of Three Instructions in a 4-Stage Pipeline

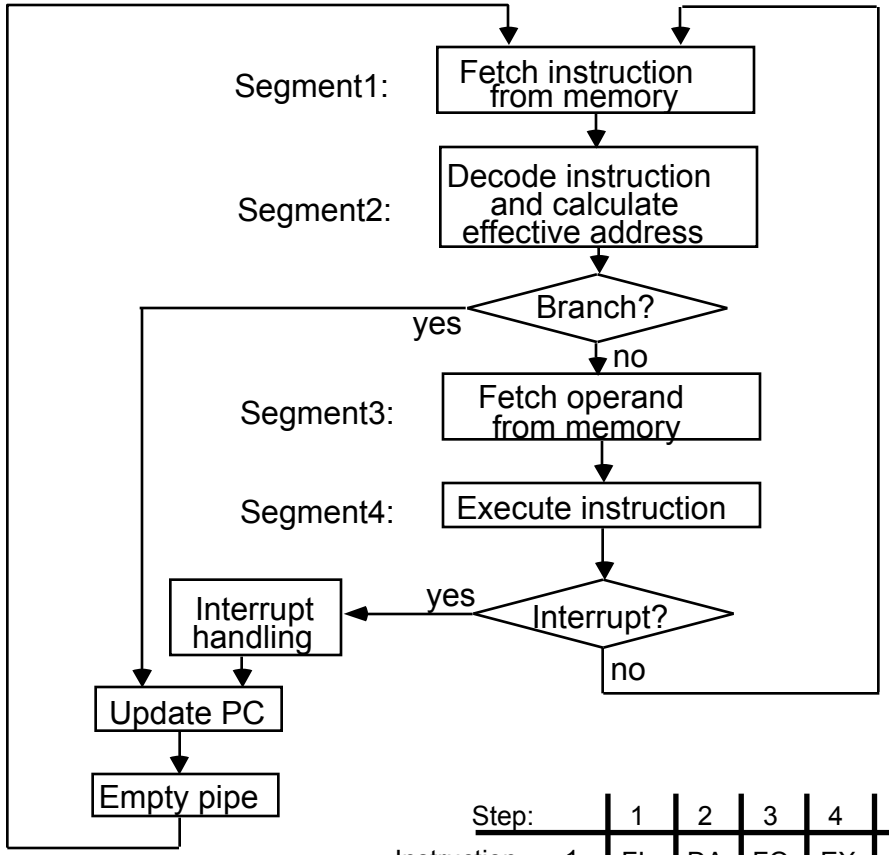
Conventional



Pipelined



INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

MAJOR HAZARDS IN PIPELINED EXECUTION

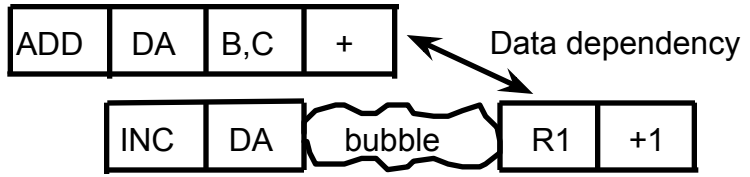
Structural hazards(Resource Conflicts)

Hardware Resources required by the instructions in simultaneous overlapped execution cannot be met

Data hazards (Data Dependency Conflicts)

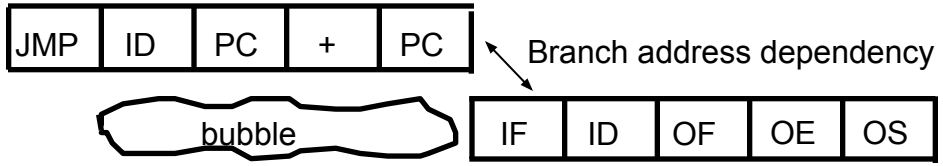
An instruction scheduled to be executed in the pipeline requires the result of a previous instruction, which is not yet available

R1 <- B + C
R1 <- R1 + 1



Control hazards (branch difficulties)

Branches and other instructions that change the PC make the fetch of the next instruction to be delayed



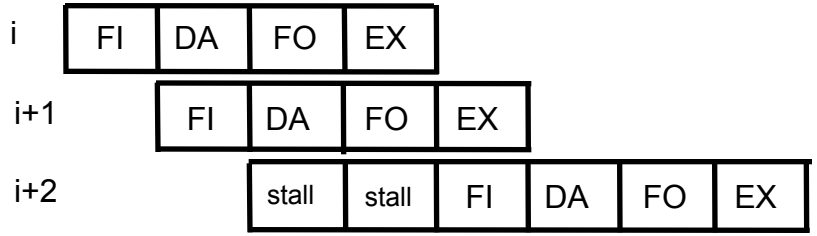
Hazards in pipelines may make it necessary to *stall* the pipeline ⇨ Pipeline Interlock: Detect Hazards Stall until it is cleared

STRUCTURAL HAZARDS

Structural Hazards

Occur when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute

Example: With one memory-port, a data and an instruction fetch cannot be initiated in the same clock



The Pipeline is stalled for a structural hazard
<- Two Loads with one port memory
-> Two-port memory will serve without stall

DATA HAZARDS

Data Hazards

Occurs when the execution of an instruction depends on the results of a previous instruction

```
ADD     R1, R2, R3
SUB     R4, R1, R5
```

Data hazard can be dealt with either hardware techniques or software technique

Hardware Technique

Interlock
- hardware detects the data dependencies and delays the scheduling of the dependent instruction by stalling enough clock cycles

Forwarding (bypassing, short-circuiting)
- Accomplished by a data path that routes a value from a source (usually an ALU) to a user, bypassing a designated register. This allows the value to be produced to be used at an earlier stage in the pipeline than would otherwise be possible

Software Technique

Instruction Scheduling(compiler) for *delayed load*

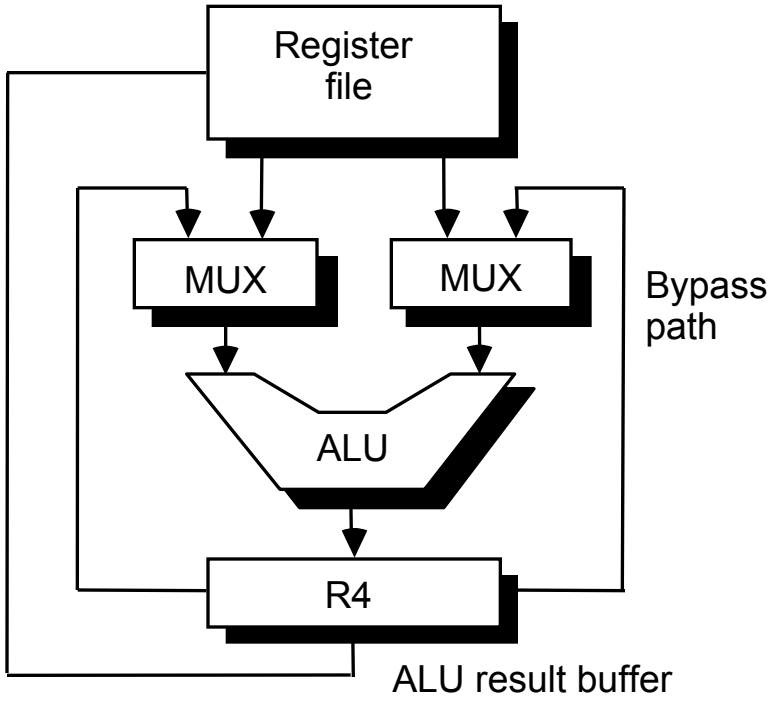
FORWARDING HARDWARE

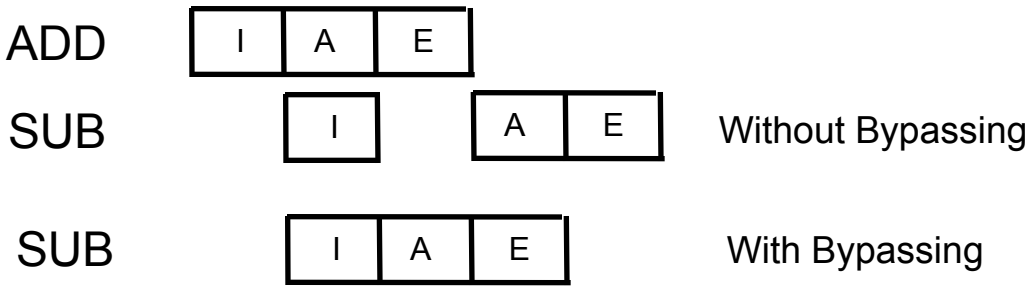
Example:

```
ADD     R1, R2, R3
SUB     R4, R1, R5
```

3-stage Pipeline

I: Instruction Fetch
A: Decode, Read Registers,
ALU Operations
E: Write the result to the
destination register





INSTRUCTION SCHEDULING TO AVOID LOAD HAZARDS

```
a = b + c;
d = e - f;
```

assume a, b, c, d ,e, and f are in memory

Unscheduled code:

```
    LW    Rb, [b]
    LW    Rc, [c]
→   ADD   Ra, Rb, Rc
→   SW    a, [Ra]
    LW    Re, [e]
→   LW    Rf, [f]
→   SUB   Rd, Re, Rf
    SW    [d], Rd
```

Scheduled Code:

```
    LW    Rb, [b]
    LW    Rc, [c]
    LW    Re, [e]
    ADD   Ra, Rb, Rc
    LW    Rf, [f]
    SW    [a], Ra
→   SUB   Rd, Re, Rf
    SW    [d], Rd
```

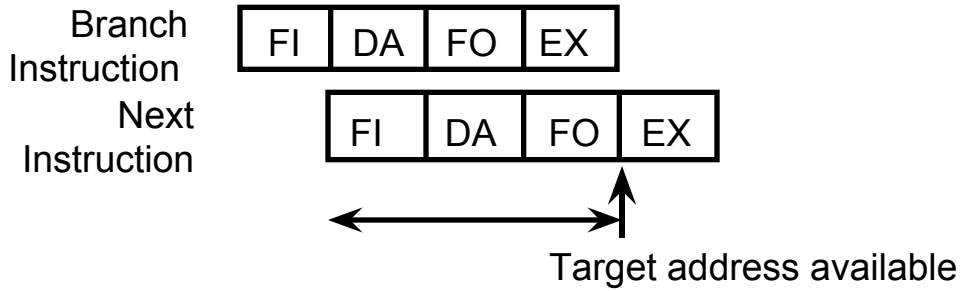
Delayed Load

A load requiring that the following instruction not use its result

CONTROL HAZARDS (branch difficulties)

Branch Instructions

- Branch target address is not known until the branch instruction is completed



- Stall -> waste of cycle times

Dealing with Control Hazards

* Prefetch Target Instruction
* Branch Target Buffer
* Loop Buffer
* Branch Prediction
* Delayed Branch

CONTROL HAZARDS

Pre fetch Target Instruction

- Fetch instructions in both streams, branch not taken and branch taken
- Both are saved until branch branch is executed. Then, select the right instruction stream and discard the wrong stream

Branch Target Buffer(BTB; Associative Memory)

- Entry: Addr of previously executed branches; Target instruction and the next few instructions
- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

Loop Buffer(High Speed Register file)

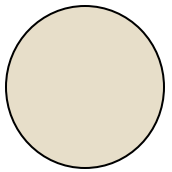
- Storage of entire loop that allows to execute a loop without accessing memory

Branch Prediction

- Guessing the branch condition, and fetch an instruction stream based on the guess. Correct guess eliminates the branch penalty

Delayed Branch

- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction



RISC PIPELINE

RISC

- Machine with a very fast clock cycle that executes at the rate of one instruction per cycle
  <- Simple Instruction Set
  Fixed Length Instruction Format
  Register-to-Register Operations

Instruction Cycles of Three-Stage Instruction Pipeline

Data Manipulation Instructions

I: Instruction Fetch
A: Decode, Read Registers, ALU Operations
E: Write a Register

Load and Store Instructions

I: Instruction Fetch
A: Decode, Evaluate Effective Address
E: Register-to-Memory or Memory-to-Register

Program Control Instructions

I: Instruction Fetch
A: Decode, Evaluate Branch Address
E: Write Register(PC)

DELAYED LOAD

LOAD: R1 ← M[address 1]
LOAD: R2 ← M[address 2]
ADD: R3 ← R1 + R2
STORE: M[address 3] ← R3

Three-segment pipeline timing

Pipeline timing with data conflict

Clock cycles:	1	2	3	4	5	6
1. Load *R*1	I	A	E			
2. Load *R*2		I	A	E		
3. Add *R*1 + *R*2			I	A	E	
4. Store *R*3				I	A	E

Pipeline timing with delayed load

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

The data dependency is taken care by the compiler rather than the hardware

DELAYED BRANCH

Compiler analyzes the instructions before and after the branch and rearranges the program sequence by inserting useful instructions in the delay steps

Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. NOP						I	A	E		
7. NOP							I	A	E	
8. Instr. in X								I	A	E

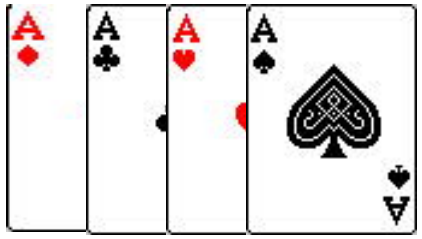
Rearranging the instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instr. in X						I	A	E

An Overview of Parallel Processing

- What is parallel processing?
  - Parallel processing is a method to improve computer system performance by executing two or more instructions simultaneously.
- The goals of parallel processing.
  - One goal is to reduce the "wall-clock" time or the amount of real time that you need to wait for a problem to be solved.
  - Another goal is to solve bigger problems that might not fit in the limited memory of a single CPU.

An Analogy of Parallelism



The task of ordering a shuffled deck of cards by suit and then by rank can be done faster if the task is carried out by two or more people. By splitting up the decks and performing the instructions simultaneously, then at the end combining the partial solutions you have performed parallel processing.

Another Analogy of Parallelism

Another analogy is having several students grade quizzes simultaneously. Quizzes are distributed to a few students and different problems are graded by each student at the same time. After they are completed, the graded quizzes are then gathered and the scores are recorded.

Parallelism in Uniprocessor Systems

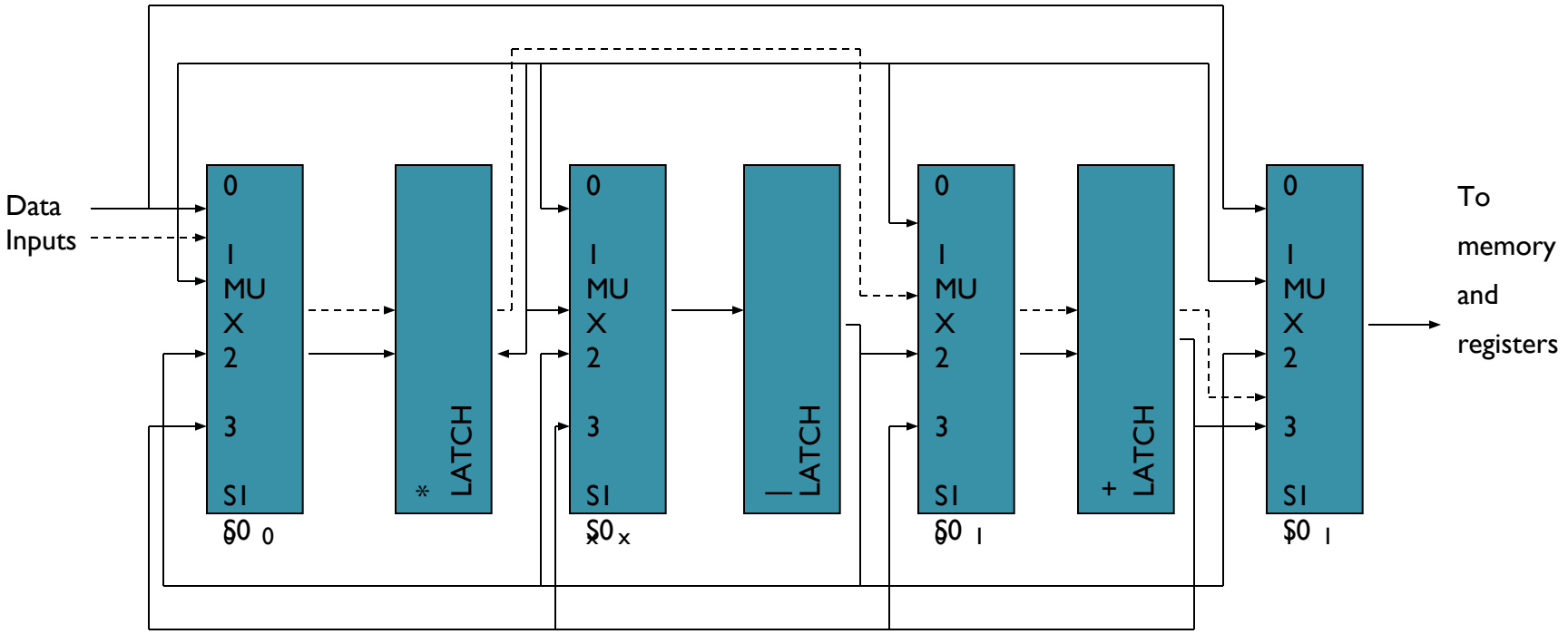
- It is possible to achieve parallelism with a uniprocessor system.
  - Some examples are the instruction pipeline, arithmetic pipeline, I/O processor.
- Note that a system that performs different operations on the same instruction is not considered parallel.
- Only if the system processes two different instructions simultaneously can it be considered parallel.

Parallelism in a Uniprocessor System

- A reconfigurable arithmetic pipeline is an example of parallelism in a uniprocessor system.

  Each stage of a reconfigurable arithmetic pipeline has a multiplexer at its input. The multiplexer may pass input data, or the data output from other stages, to the stage inputs. The control unit of the CPU sets the select signals of the multiplexer to control the flow of data, thus configuring the pipeline.

A Reconfigurable Pipeline With Data Flow for the Computation
A[i] □ B[i] * C[i] + D[i]



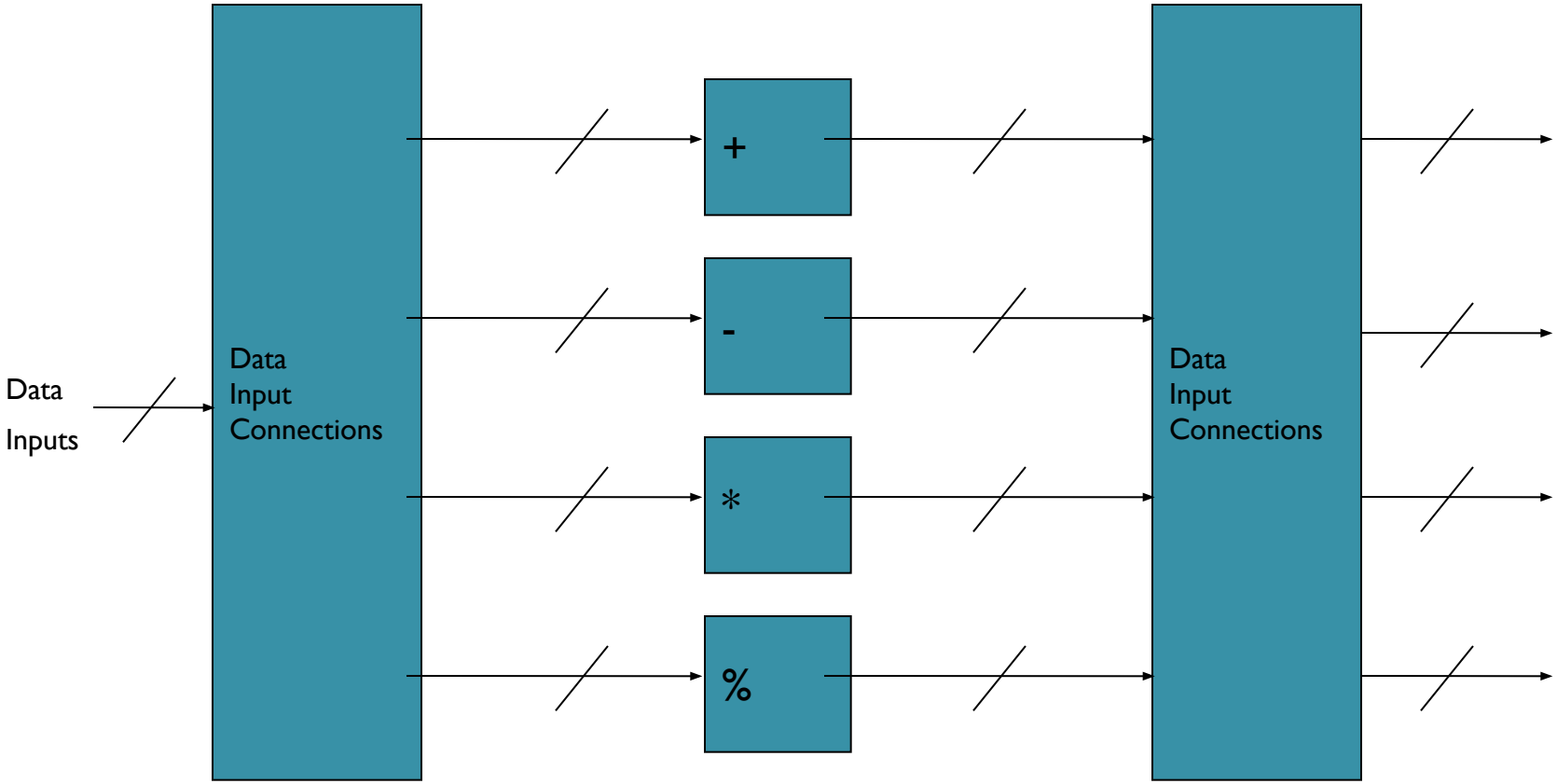
Although arithmetic pipelines can perform many iterations of the same operation in parallel, they cannot perform different operations simultaneously. To perform different arithmetic operations in parallel, a CPU may include a vectored arithmetic unit.

Vector Arithmetic Unit

A vector arithmetic unit contains multiple functional units that perform addition, subtraction, and other functions. The control unit routes input values to the different functional units to allow the CPU to execute multiple instructions simultaneously.

For the operations A□B+C and D□E-F, the CPU would route B and C to an adder and then route E and F to a subtractor for simultaneous execution.

A Vectored Arithmetic Unit



A□B+C

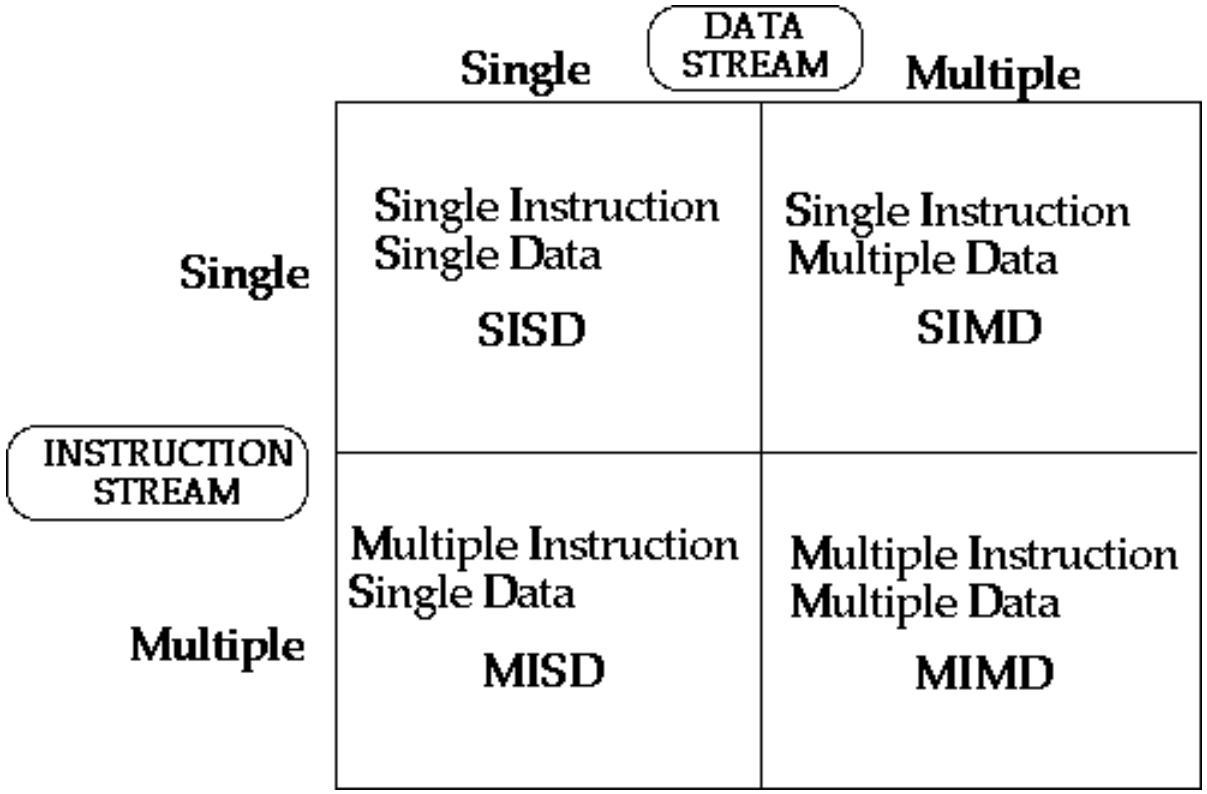
D□E-F

Organization of Multiprocessor Systems

- 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.

Taxonomy of Computer Architectures

Simple Diagrammatic Representation



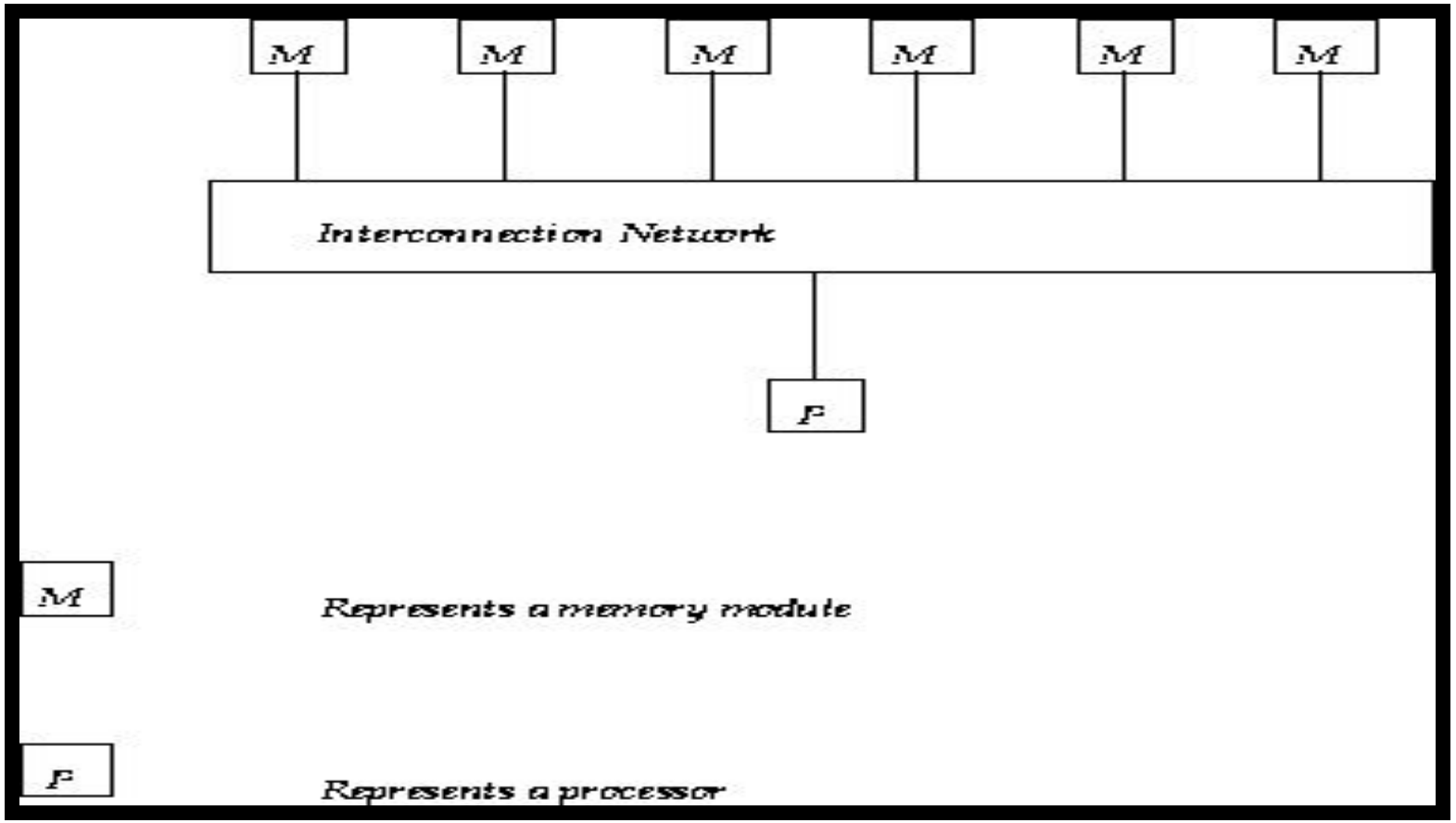
4 categories of Flynn's classification of multiprocessor systems by their instruction and data streams

Single Instruction, Single Data (SISD)

- SISD machines executes a single instruction on individual data values using a single processor.
- Based on traditional Von Neumann uniprocessor architecture, instructions are executed sequentially or serially, one step after the next.
- Until most recently, most computers are of SISD type.

SISD

Simple Diagrammatic Representation

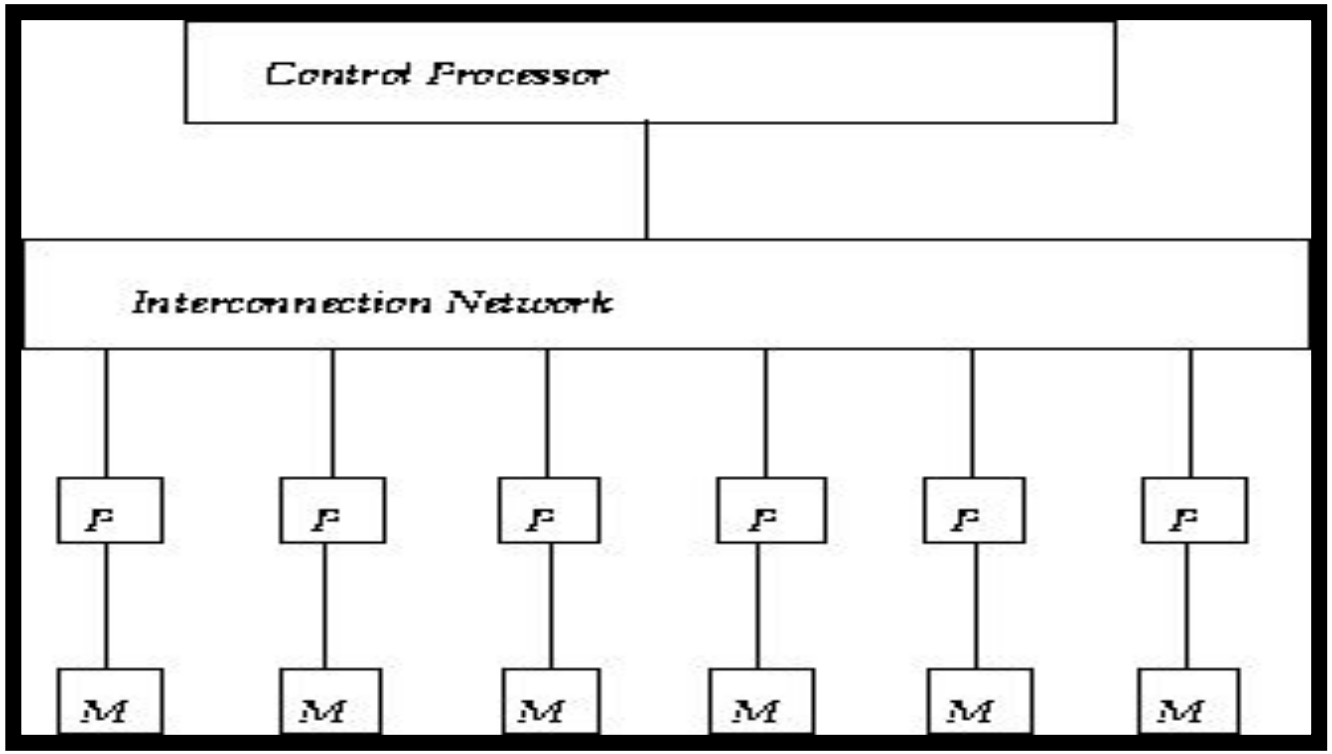


Single Instruction, Multiple Data (SIMD)

- An SIMD machine executes a single instruction on multiple data values simultaneously using many processors.
- Since there is only one instruction, each processor does not have to fetch and decode each instruction. Instead, a single control unit does the fetch and decoding for all processors.
- SIMD architectures include array processors.

SIMD

Simple Diagrammatic Representation



Multiple Instruction, Single Data (MISD)

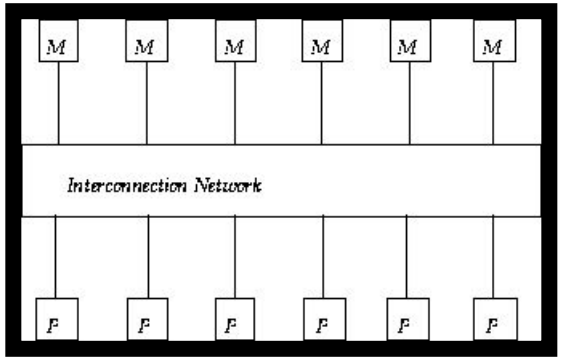
- This category does not actually exist. This category was included in the taxonomy for the sake of completeness.

Multiple Instruction, Multiple Data (MIMD)

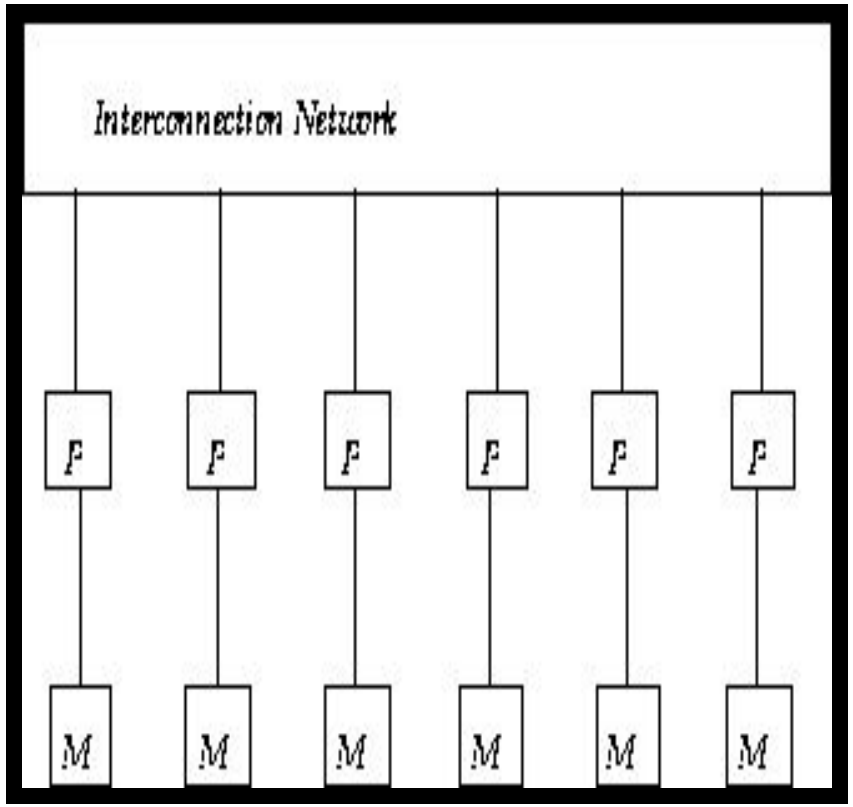
- MIMD machines are usually referred to as multiprocessors or multicomputers.
- It may execute multiple instructions simultaneously, contrary to SIMD machines.
- Each processor must include its own control unit that will assign to the processors parts of a task or a separate task.
- It has two subclasses: Shared memory and distributed memory

MIMD

Simple Diagrammatic Representation (Shared Memory)



Simple Diagrammatic Representation(DistributedMemory)



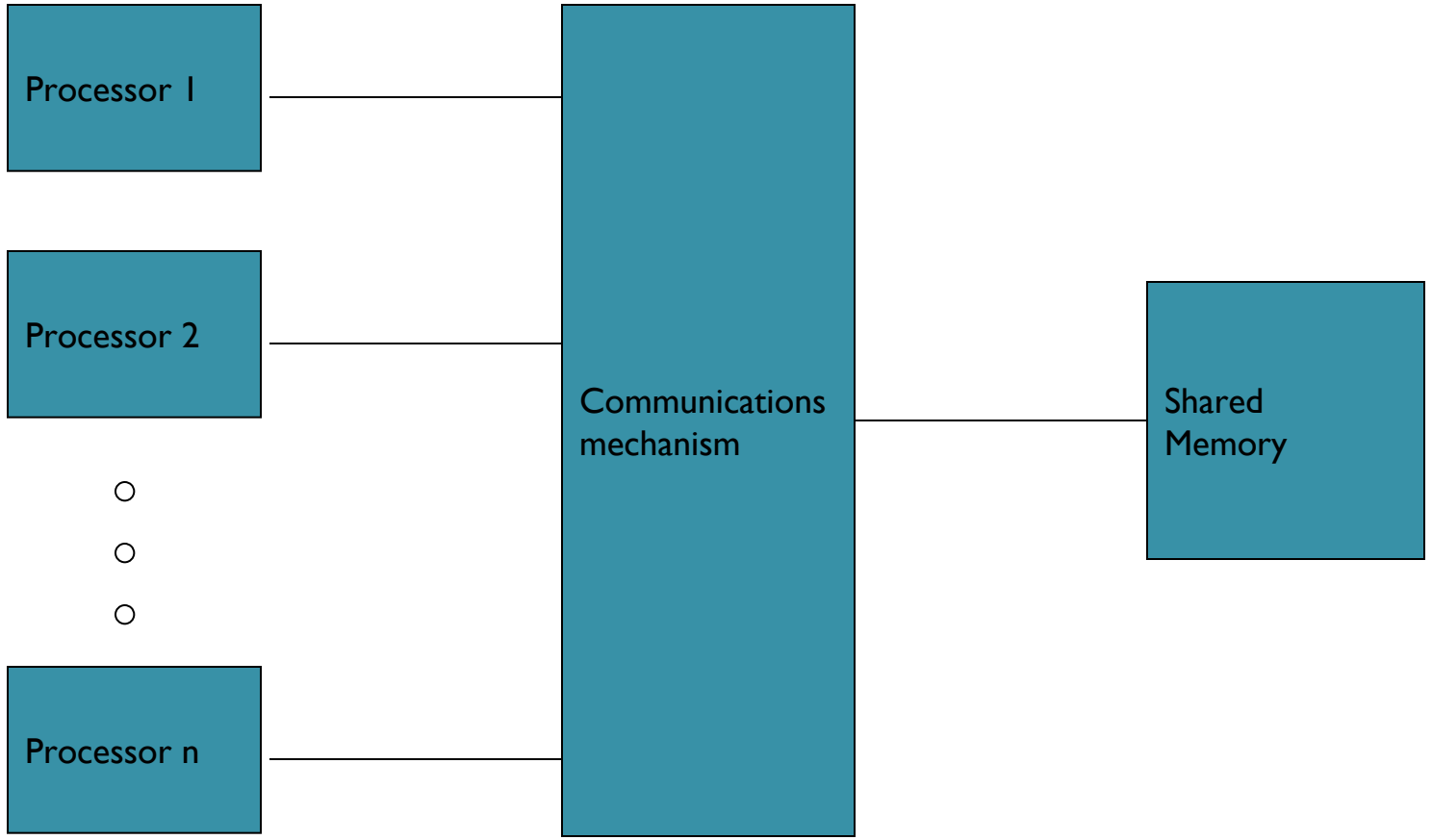
MIMD System Architectures

- Finally, the architecture of a MIMD system, contrast to its topology, refers to its connections to its system memory.
- A systems may also be classified by their architectures. Two of these are:
  - Uniform memory access (UMA)
  - Nonuniform memory access (NUMA)

Uniform memory access (UMA)

- The UMA is a type of symmetric multiprocessor, or SMP, that has two or more processors that perform symmetric functions. UMA gives all CPUs equal (uniform) access to all memory locations in shared memory. They interact with shared memory by some communications mechanism like a simple bus or a complex multistage interconnection network.

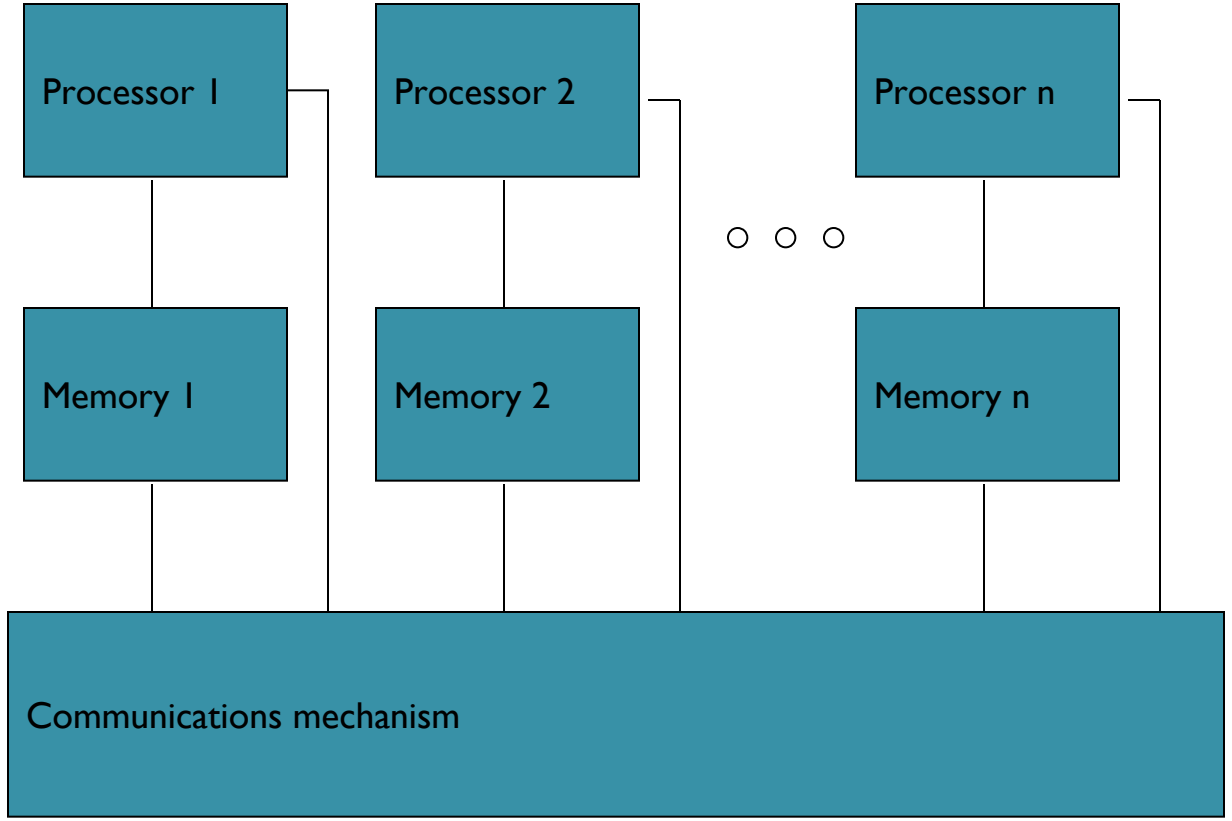
Uniform memory access (UMA) Architecture



Nonuniform memory access (NUMA)

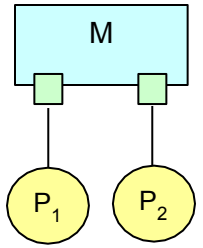
- NUMA architectures, unlike UMA architectures do not allow uniform access to all shared memory locations. This architecture still allows all processors to access all shared memory locations but in a nonuniform way, each processor can access its local shared memory more quickly than the other memory modules not next to it.

Nonuniform memory access (NUMA) Architecture



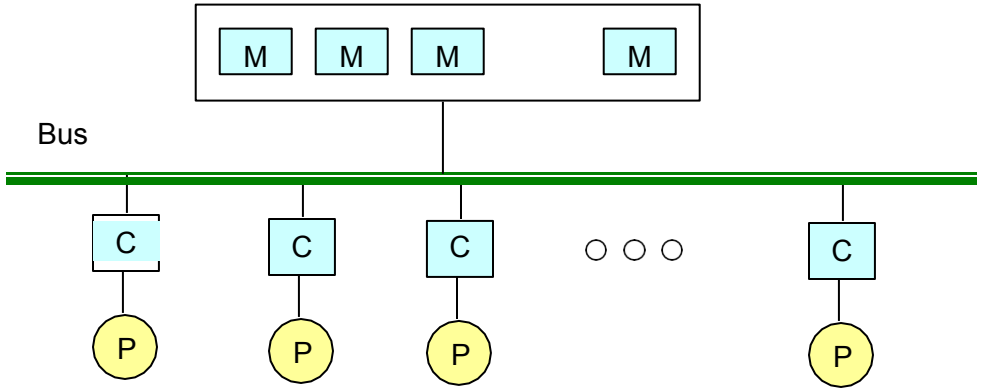
Classification of Shared Memory Systems

- Shared memory systems are multi-port and categorized as follows:
  - Uniform memory Access (UMA)
  - Non-uniform Memory Access (NUMA)
  - Cache Only Memory Architecture (COMA)



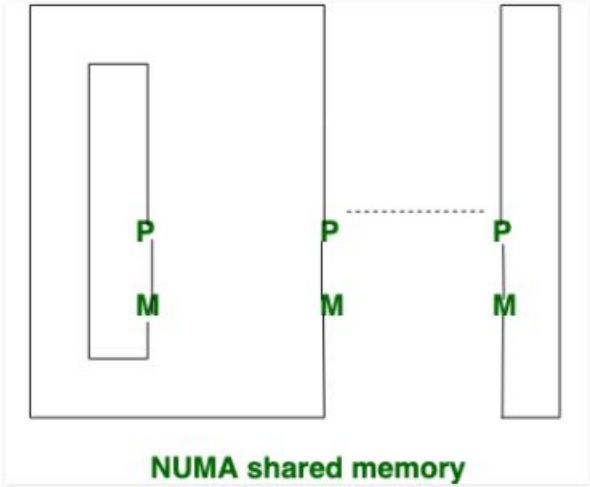
Classification of Shared Memory Systems

- Uniform Memory Access (UMA)
  - Shared memory is accessible by all processors through an interconnection network in the same way a single processor accesses its memory.
  - All processors have equal access time to any memory location.
  - Because access to the shared memory is balanced, these systems are called SMP (symmetric multiprocessor)



Classification of Shared Memory Systems

- Non Uniform Memory Access (NUMA)
  - Each processor has part of the shared memory attached.
  - The access time to modules depend on the distance to the processor, which results a non-uniform memory access time.

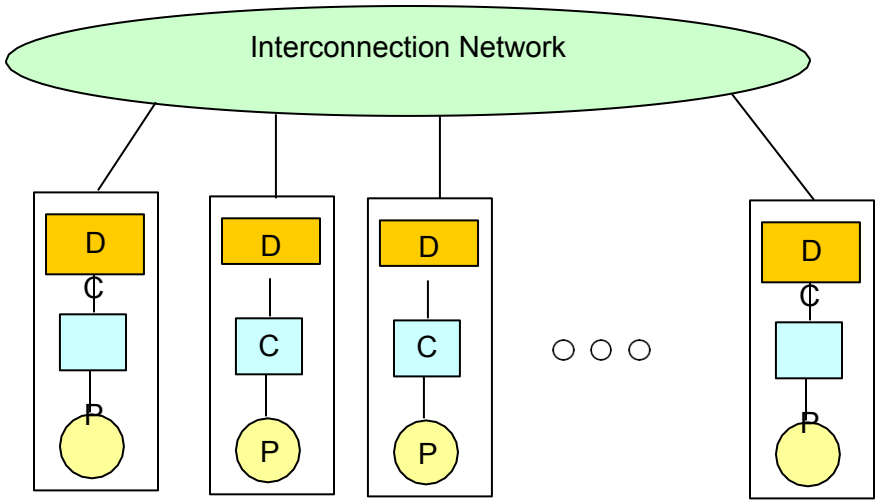


Difference between UMA and NUMA

S.NO	UMA	NUMA
1.	UMA stands for Uniform Memory Access.	NUMA stands for Non-uniform Memory Access.
2.	In Uniform Memory Access, Single memory controller is used.	In Non-uniform Memory Access, Different memory controller is used.
3.	Uniform Memory Access is slower than non-uniform Memory Access.	Non-uniform Memory Access is faster than uniform Memory Access.
4.	Uniform Memory Access has limited bandwidth.	Non-uniform Memory Access has more bandwidth than uniform Memory Access.
5.	Uniform Memory Access is applicable for general purpose applications and time-sharing applications.	Non-uniform Memory Access is applicable for real-time applications and time-critical applications.
6.	In uniform Memory Access, memory access time is balanced or equal.	In non-uniform Memory Access, memory access time is not equal.
7.	There are 3 types of buses used in uniform Memory Access which are: Single, Multiple and Crossbar.	While in non-uniform Memory Access, There are 2 types of buses used which are: Tree and hierarchical.

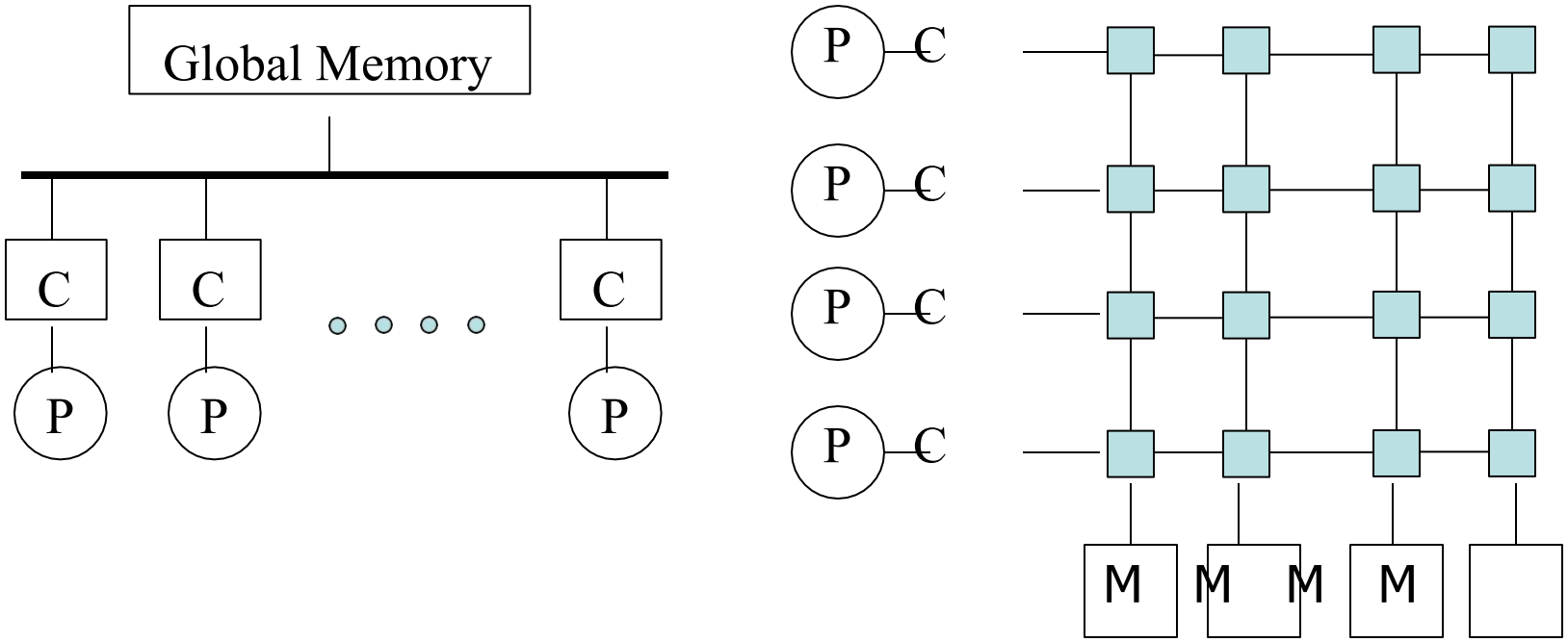
Classification of Shared Memory Systems

- Cache-Only Memory Architecture (COMA)
  - Like NUMA, each processor has part of the shared memory in COMA.
  - COMA requires the data to be migrated to the processor requesting it.



Classification of Shared Memory Systems

- Shared memory systems can be designed using bus-based or switch-based interconnection networks.



Classification of Shared Memory Systems

- Cache-memory coherence using two policies:
  - Write-Through:
    - In the write-through policy, both cache and main memory are updated with every write operation.
    - Write-Back:
    - In the write-back policy, only the cache is updated and the location is marked so that it can be copied later into main memory. .

Three-processor configuration with private caches

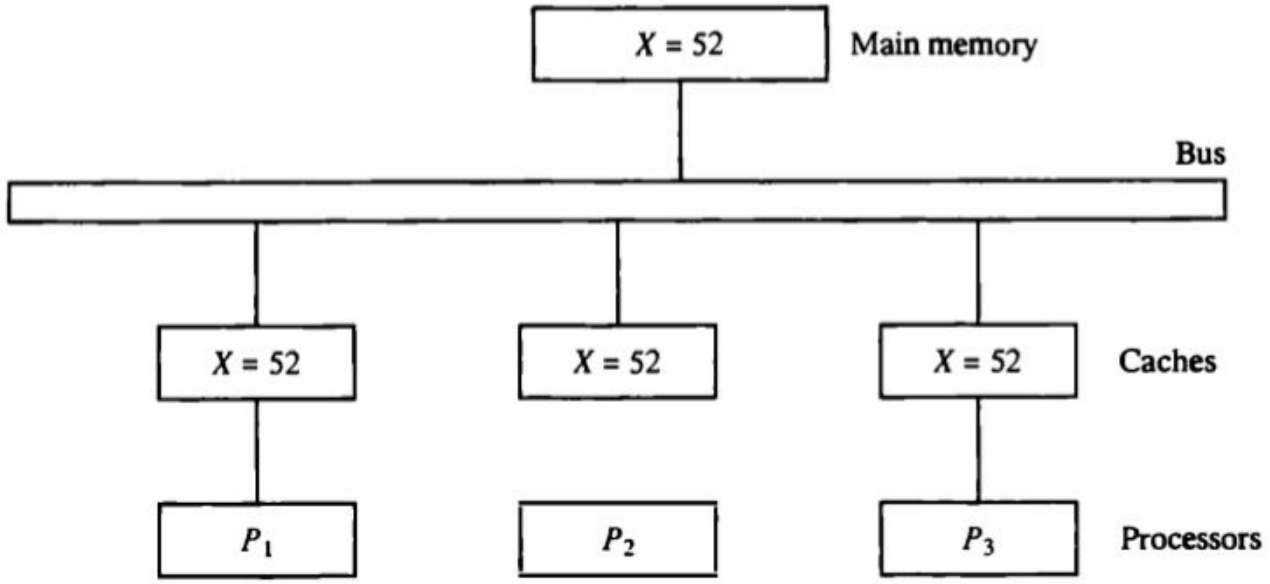
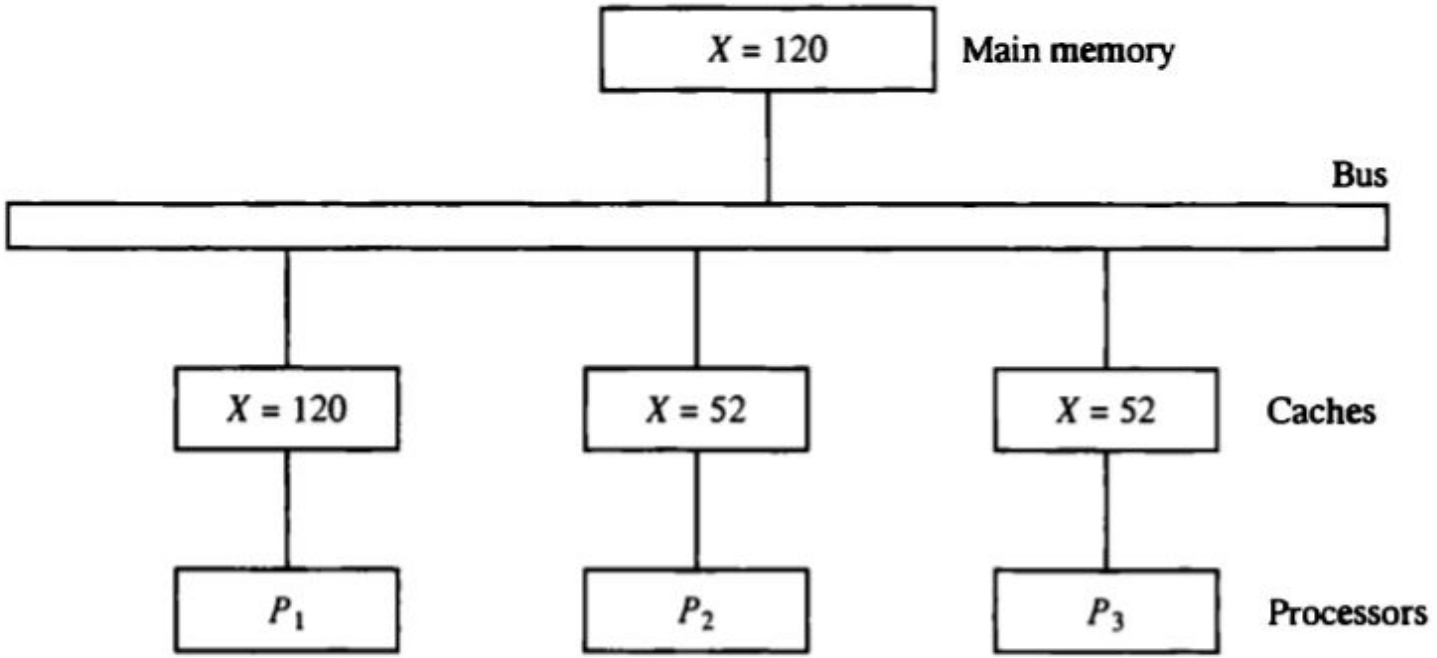


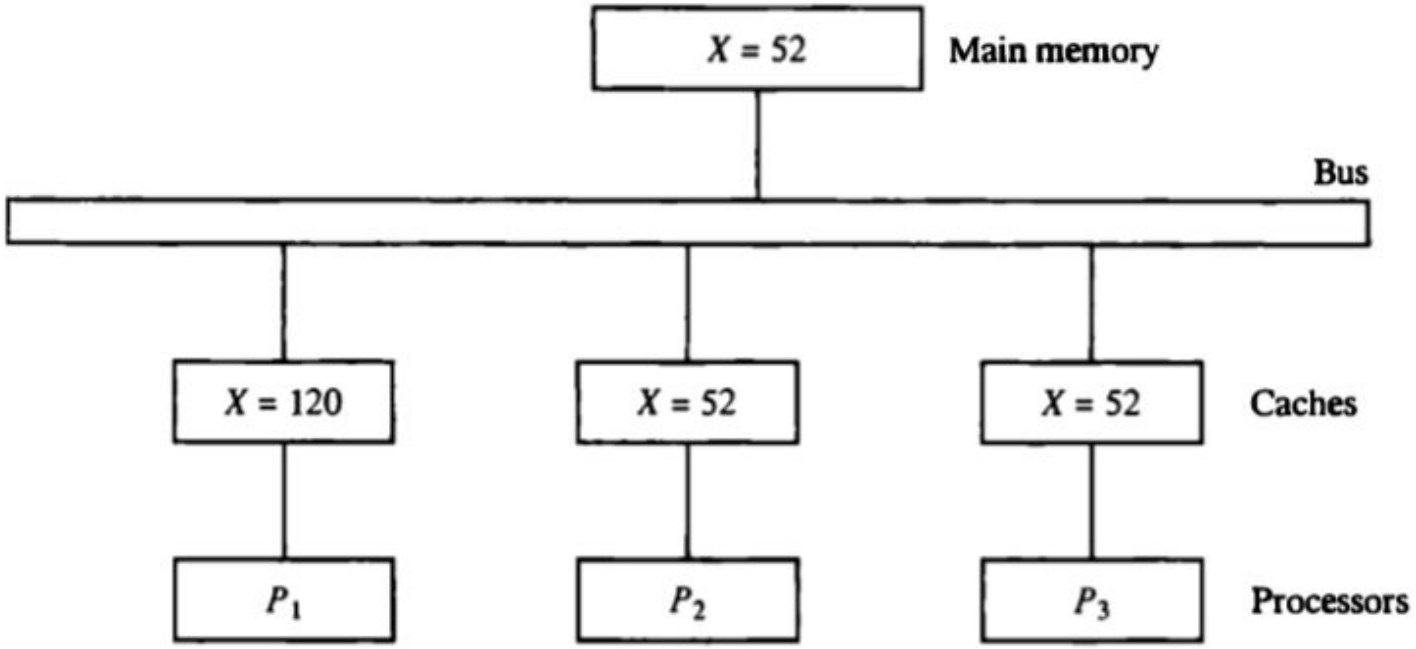
Figure 13-12 Cache configuration after a load on X.

Write through cache policy



(a) With write-through cache policy

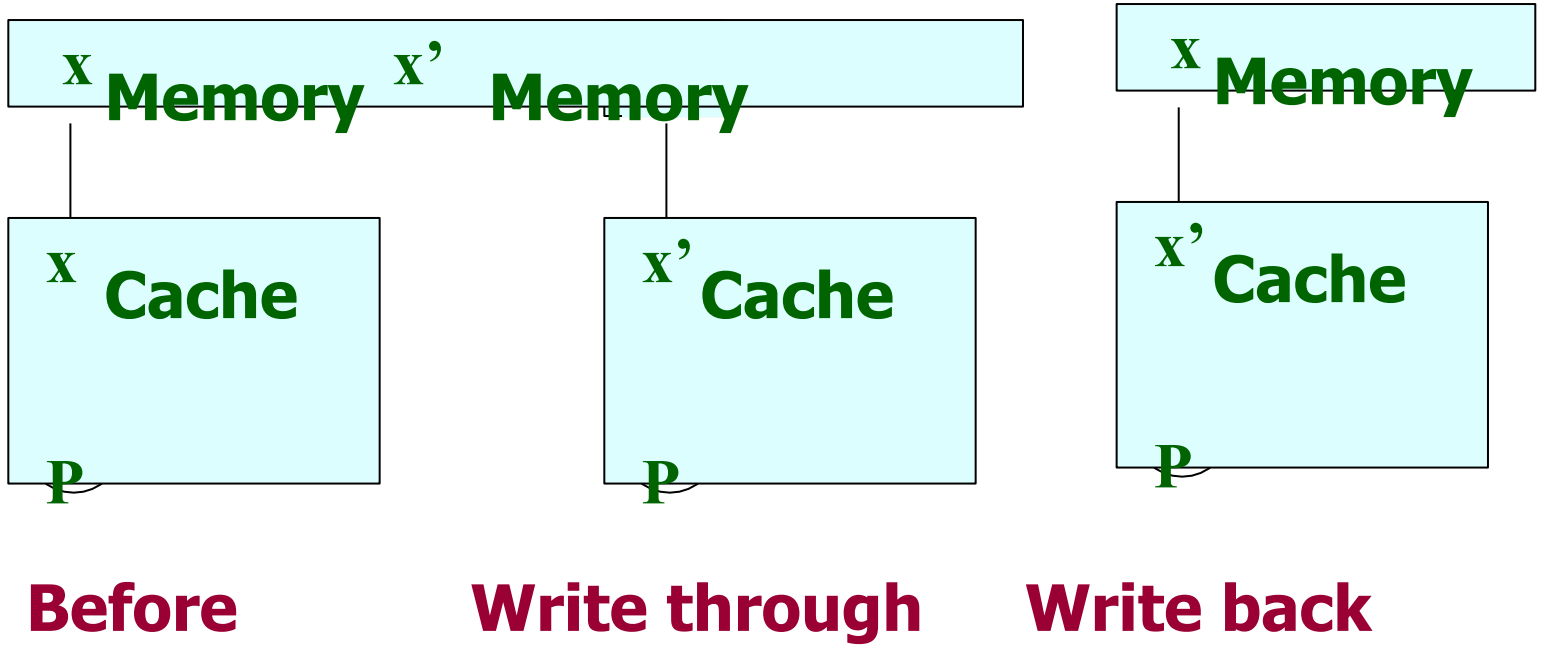
Write back cache policy



(b) With write-back cache policy

Classification of Shared Memory Systems

- Cache-memory coherence using two policies:

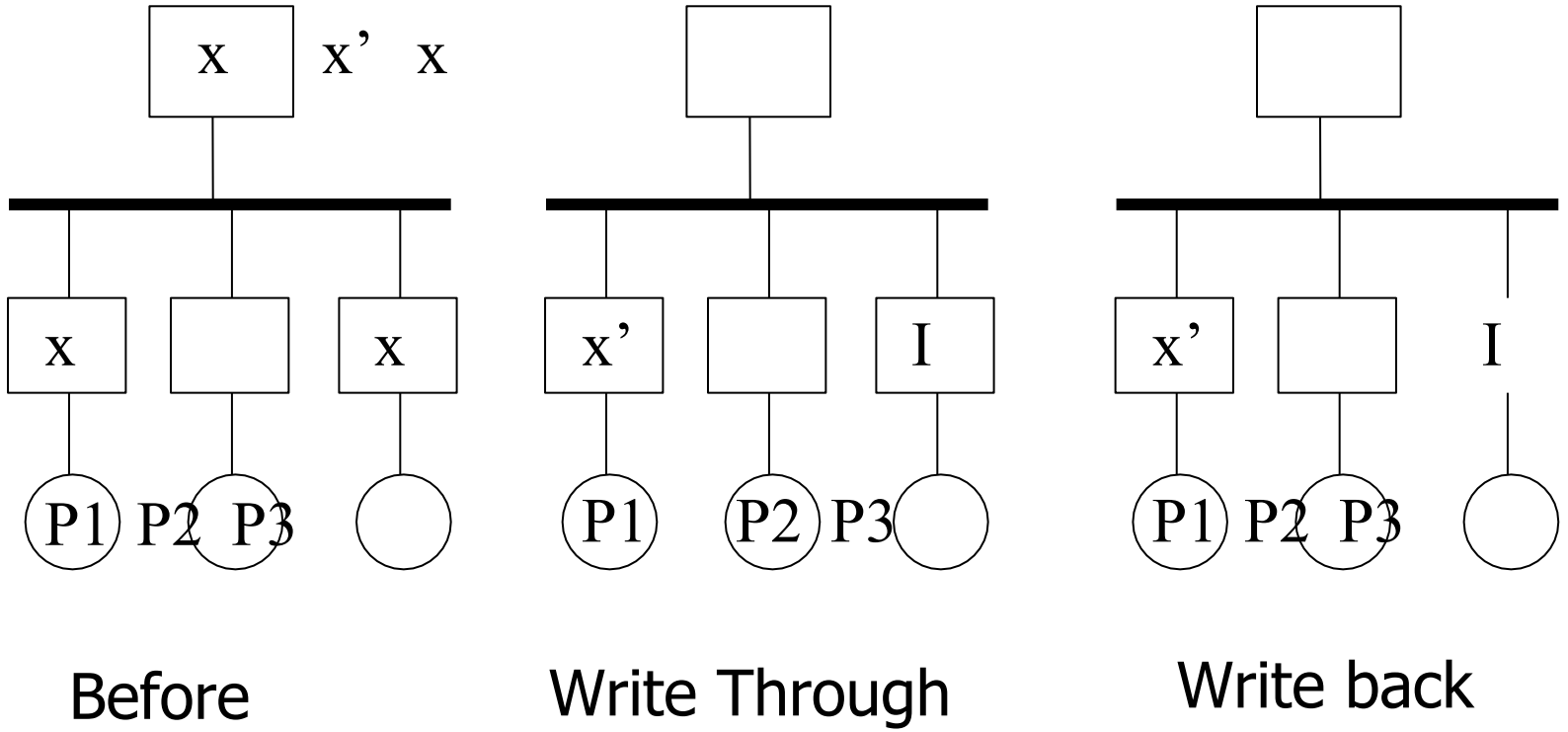


Classification of Shared Memory Systems

- Cache-cache coherence using two policies:
  - Write-Invalidate:
    - Maintains consistency by reading from local caches until a write occurs.
  - Write Update:
    - Maintains consistency by immediately updating all copies in all caches.

Classification of Shared Memory Systems

- Write-Invalidate:



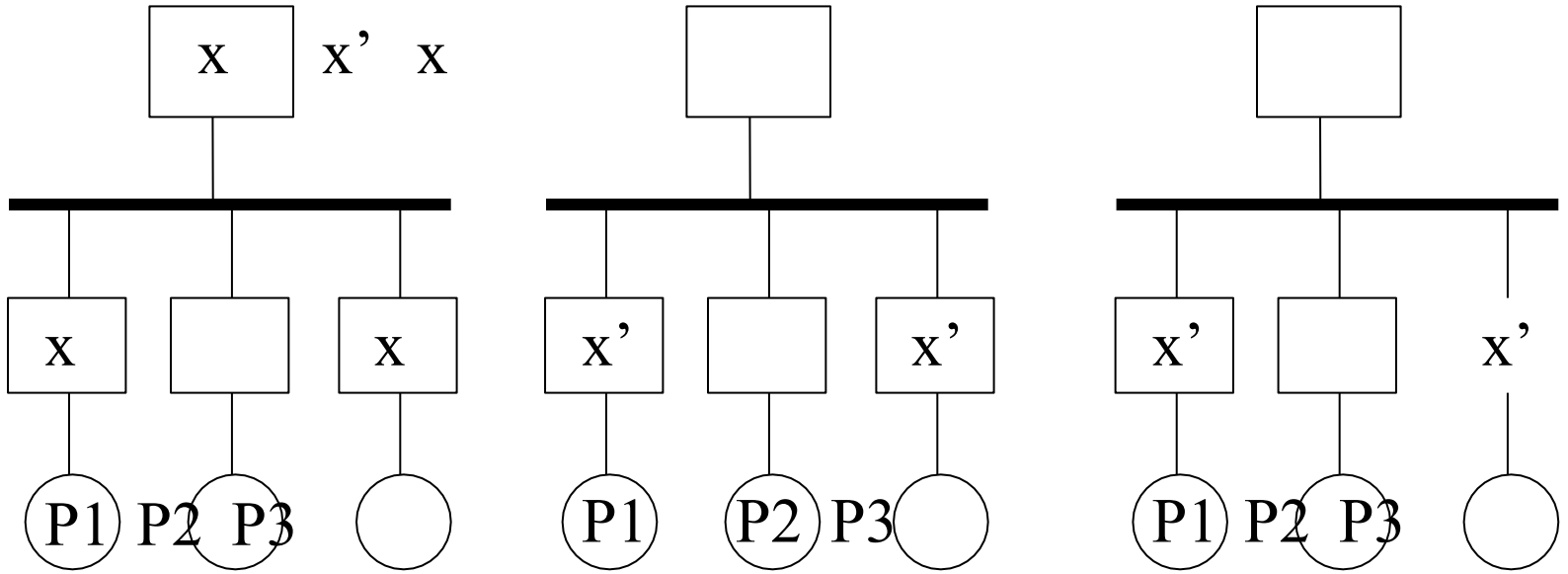
Before

Write Through

Write back

Classification of Shared Memory Systems

- Write-Update:



Before

Write Through

Write back

Classification of Shared Memory Systems

- Shared Memory System Coherence :
  - Four combinations to maintain coherence among all caches and global memory:
    - Write-Update and Write-Through.
    - Write-Update and Write-Back.
    - Write-Invalidate and Write-Through.
    - Write-Invalidate and Write-Back.