Parallel Processing and Multiprocessors

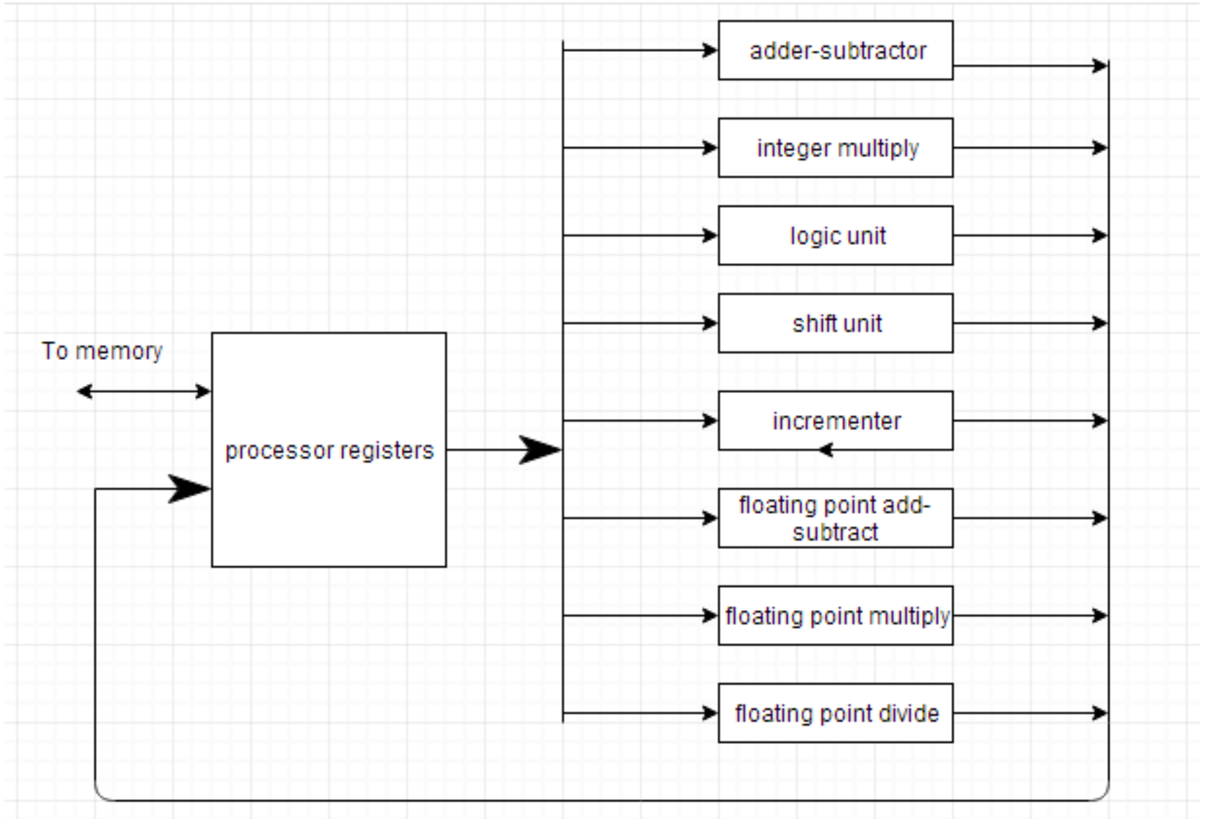
Dr. Raahat Devender Singh
CSED, TIET

In this chapter, we will study

Parallel Processing

A term used to denote a large class of techniques that are used to provide **simultaneous data-processing tasks** to **increase the computational speed of a computer system**.

Example of a Processor With Multiple Functional Units



Classification of Parallel Processing

Parallel processing can be classified based on:

>> the internal organization of the processors

>> interconnection structure between processors

>> flow of information through the system

Flynn's Classification of Parallel Processing

Flynn's classification divides computers into four major groups:

>> Single instruction stream, single data stream (SISD)
>> Single instruction stream, multiple data stream (SIMD)
>> Multiple instruction stream, single data stream (MISD)
>> Multiple instruction stream, multiple data stream (MIMD)

Instruction stream: Sequence of instructions read from memory
Data stream: Operations performed on the data in the processor

Pipelining

Pipelining

Pipelining is a technique of decomposing a sequential process into sub-operations, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments.

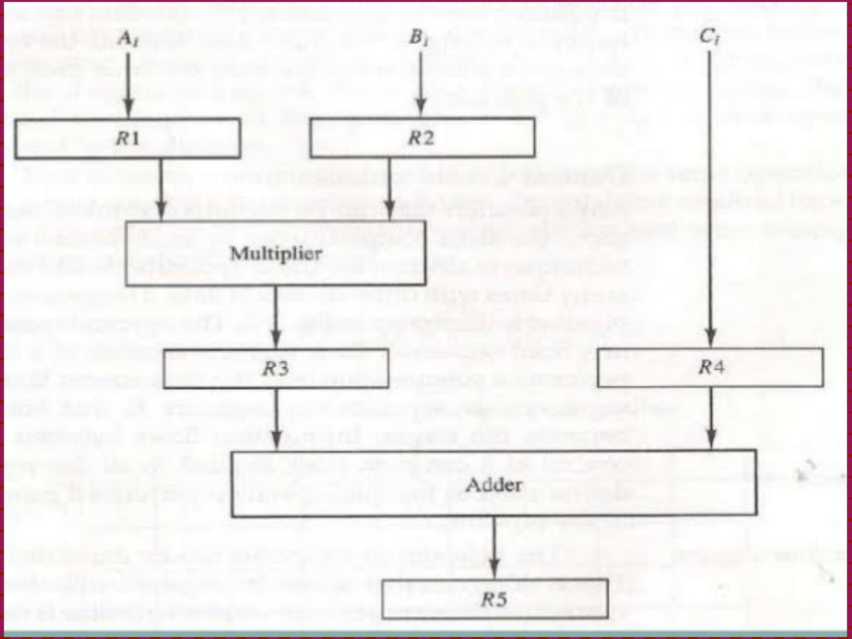
Example of Pipeline Organization

Say we want to perform combined multiplication and addition operation with a stream of numbers…

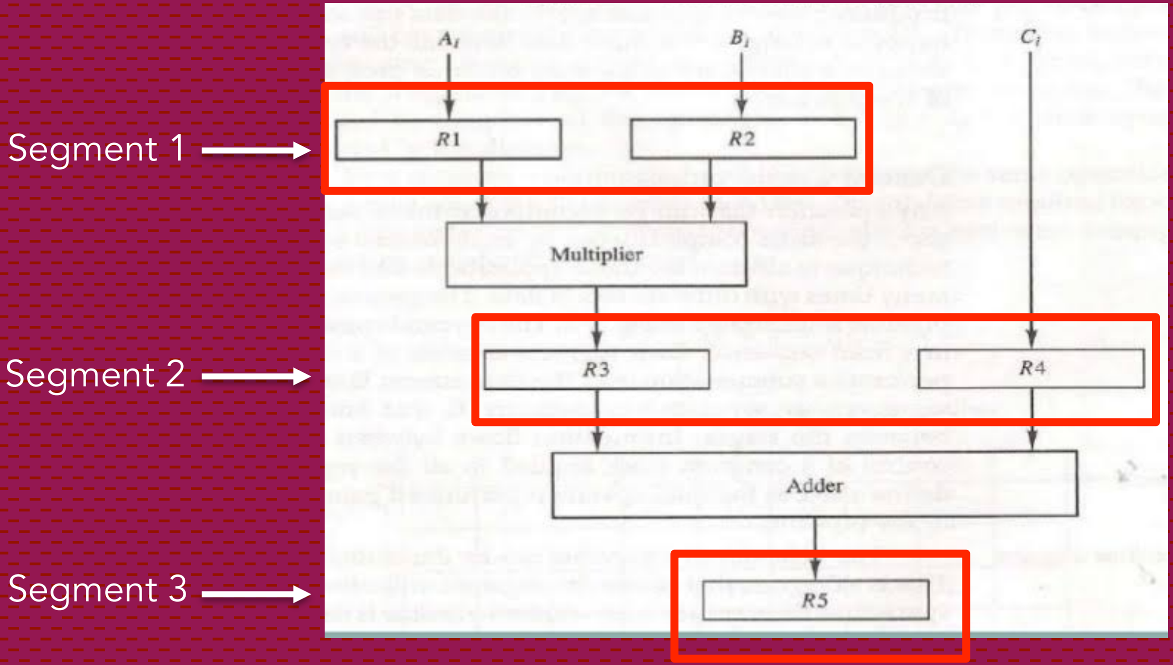
$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots., 7$$

Each sub-operation is to be implemented in a segment within a pipeline; each segment has one or two registers and a combinational circuit.

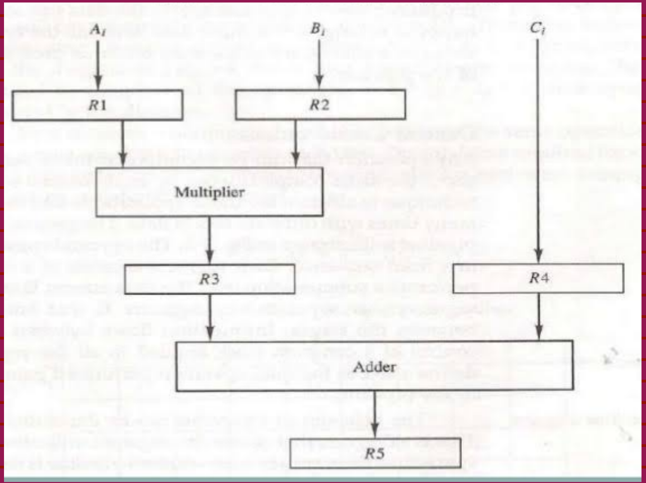
Example of Pipeline Organization



Example of Pipeline Organization



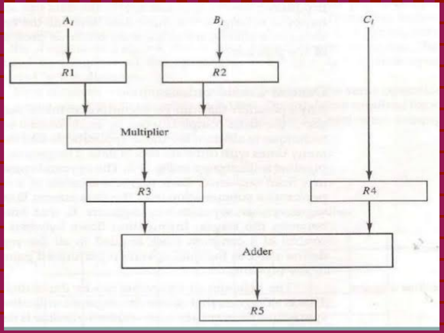
Example of Pipeline Organization

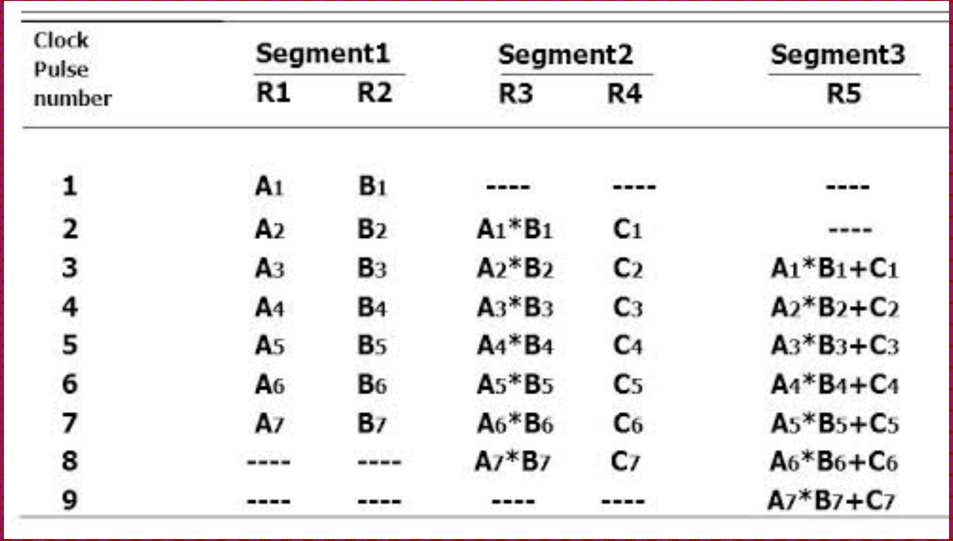


R1 to R5 are registers that receive new data with every clock pulse;
multipler and adder are combinational cicuits.
The sub-operations performed in each segment are as follows:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow Ci$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

Example of Pipeline Organization





Clock Pulse number	Segment1		Segment2		Segment3
	R1	R2	R3	R4	R5
1	A_1	B_1	----	----	----
2	A_2	B_2	A_1*B_1	C_1	----
3	A_3	B_3	A_2*B_2	C_2	$A_1*B_1+C_1$
4	A_4	B_4	A_3*B_3	C_3	$A_2*B_2+C_2$
5	A_5	B_5	A_4*B_4	C_4	$A_3*B_3+C_3$
6	A_6	B_6	A_5*B_5	C_5	$A_4*B_4+C_4$
7	A_7	B_7	A_6*B_6	C_6	$A_5*B_5+C_5$
8	----	----	A_7*B_7	C_7	$A_6*B_6+C_6$
9	----	----	----	----	$A_7*B_7+C_7$

Contents of Registers

Pipeline Organization: General Considerations

Pipelining is efficient in those applications that need to repeat the same task many times with different sets of data.

Pipeline Organization: General Considerations

Speedup: Consider a case where a *k-segment* pipeline with a clock cycle time t_p is used to execute *n* tasks.

The first task requires time kt_p to complete its operation (since there are k segments in the pipeline).

The remaining $n - 1$ tasks emerge from the pipeine at the rate of one per clock cycle and they will be completed after time equal to $(n - 1)\ t_p$.

Therefore, completion of n tasks in a k-segment pipeline requires k + (n – 1) clock cycles.

Pipeline Organization: General Considerations

Speedup: In our current example, the time required to complete all operations is 3 + (7 – 1) = 9 clock cycles.

Without pipelining, the same operation would have taken 7 * 3 = 21 clock cycles (7 tasks and 3 clock cycles for each task).

Pipeline Organization: General Considerations

Speedup Ratio: The speedup of a pipeline processor over an equivalent non-pipeline processor is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Where nt_n is the total time reqired by a non-pipeline unit to complete *n* tasks where each task takes time t_n to complete.

Pipeline Organization: General Considerations

Speedup Ratio: As the number of tasks increase, *n* become much larger than *k – 1*, and *k + n – 1* approaches the value of *n*. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, we will have $t_n = kt_p$. Speedup then becomes

$$S = k$$

This shows that the maximum theoretical speedup that a pipeline can provide is equal to the number of segements in the pipeline.

Pipeline Organization: General Considerations

In order to duplicate the theoretical speedup advantage of a pipeline process, it is necessary to construct *k identical units* that will operate in parallel.

The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent non-pipeline circuit under equal operating conditions.

Types of Pipelines

Arithmetic Pipelines

Instruction Pipelines

Arithmetic Pipeline

>> *Found in very high speed computers.*

>> *Used to implement floating-point operations, multiplication of fixed-point numbers, and scientific problems.*

Arithmetic Pipeline

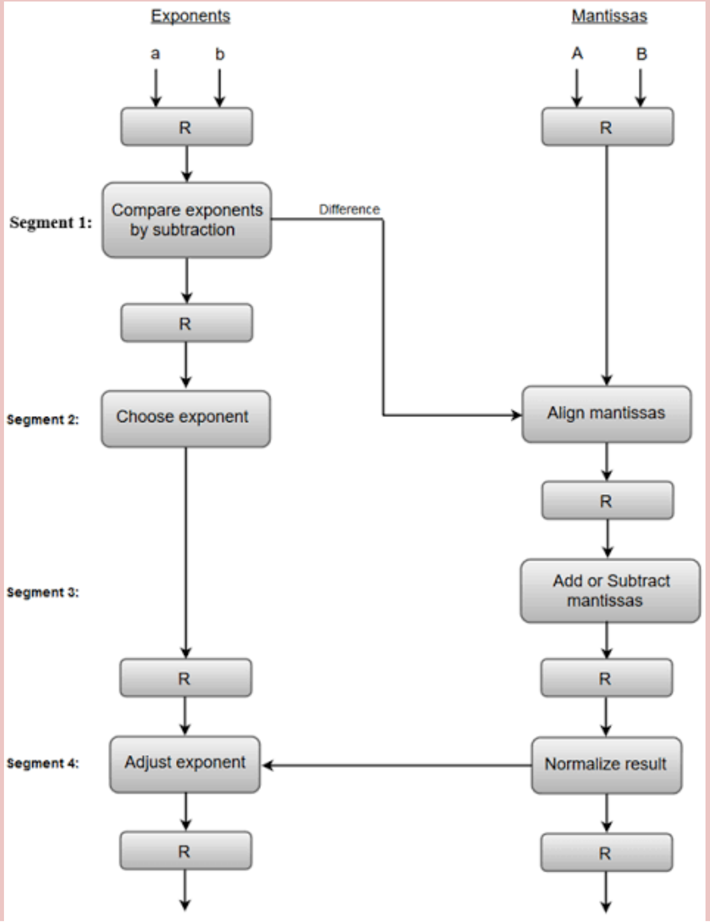
Let's consider floating-point addition and subtraction.

Inputs to a floating-point adder pipeline are two normalized floating-point binary numbers:

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

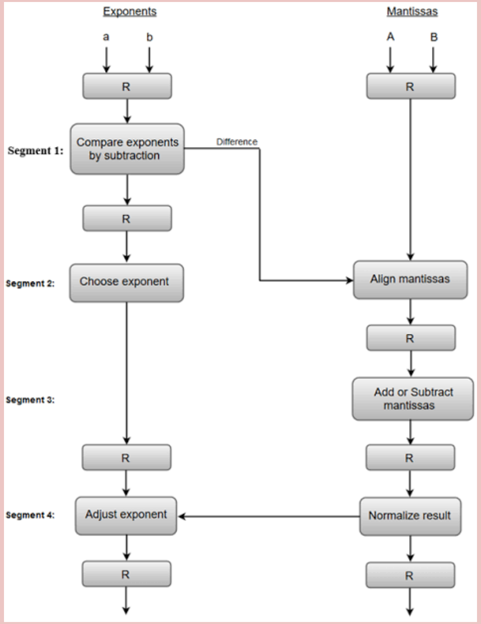
This operation can be performed using a 4-segment pipeline…



Arithmetic Pipeline:

Numerical Example

Please note that for simplicity, decimal numbers are considered.



$X = 0.9504 \times 10^3$

$Y = 0.8200 \times 10^2$

In **Segment 1**, the two exponents are subtracted to obtain 3 – 2 = 1.

The larger exponent 3 is chosen as the exponent of the result.

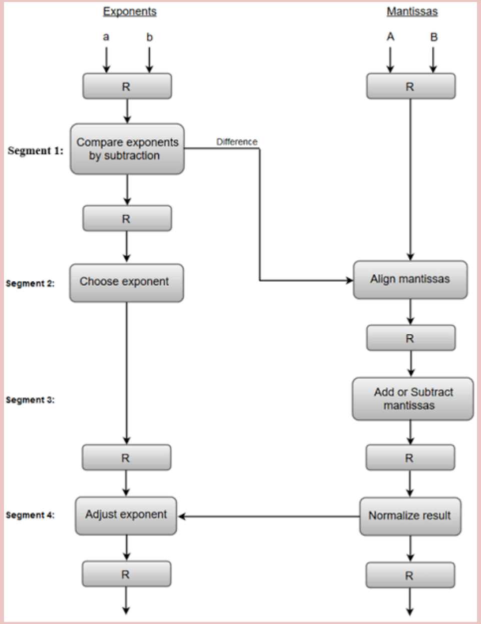
Segment 2 then shifts the mantissa of Y to the right to obtain

$X = 0.9504 \times 10^3$

$Y = 0.0820 \times 10^3$

Arithmetic Pipeline:

Numerical Example



In **Segment 3**, the addition of the two mantissas produces the sum

$$Z = 1.0324 \times 10^3$$

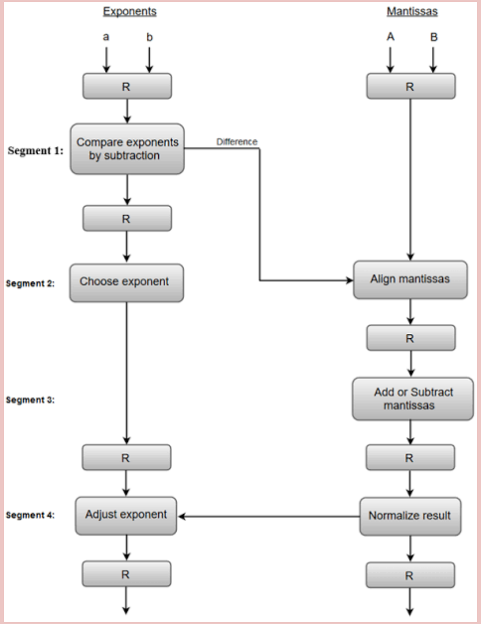
In **Segment 4**, the sum is adjusted by normalizing the result so that it has a fraction with a non-zero first digit.

This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain

$$Z = 0.10324 \times 10^4$$

Arithmetic Pipeline:

Numerical Example



Suppose the time delays of the four segments are

$t_1 = 60$ ns $t_2 = 70$ ns

$t_3 = 100$ ns $t_4 = 80$ ns

and the interface registers have a delay of $t_r = 10$ ns.

The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns.

An equivalent non-pipeline floating-point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns.

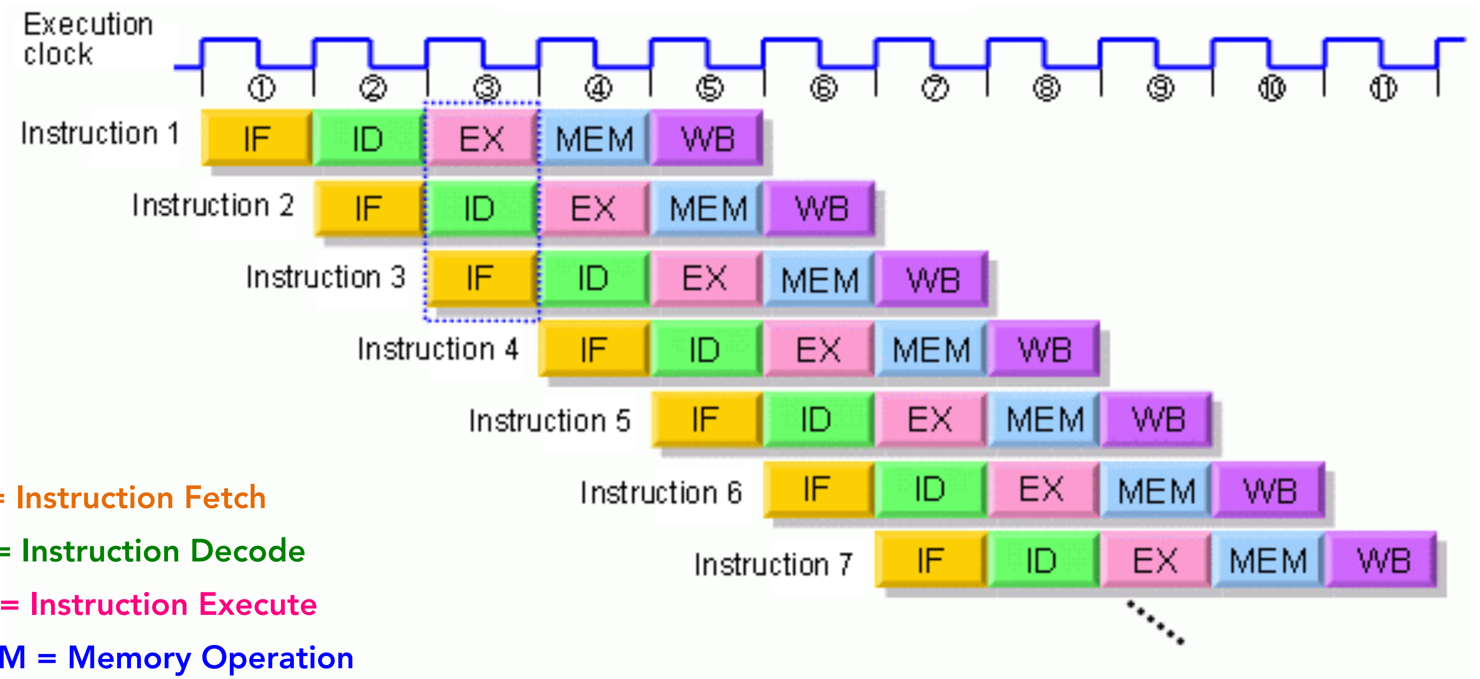
The pipeline adder-subtractor will thus have a speedup of 320/110 = 2.9 over the non-pipeline adder-subtractor.

Instruction Pipeline

>> *Reads consecutive instructions from memory while previous instructions are being executed in other segments.*

>> *This causes instruction fetch and execute phases to overlap and perform simulataneous operations.*

5-Stage Instruction Pipeline



Execution clock
① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪
Instruction 1 IF ID EX MEM WB
Instruction 2 IF ID EX MEM WB
Instruction 3 IF ID EX MEM WB
Instruction 4 IF ID EX MEM WB
Instruction 5 IF ID EX MEM WB
Instruction 6 IF ID EX MEM WB
Instruction 7 IF ID EX MEM WB

IF = Instruction Fetch

ID = Instruction Decode

EX = Instruction Execute

MEM = Memory Operation

WB = Write Back

Instruction Pipeline: Sequence of Steps

1. Fetch instruction from memory
2. Decode instruction
3. Calculate effective address
4. Fetch operand(s) from memory
5. Execute instruction
6. Store result in a proper place

Instruction Pipeline: Difficulties

Factors that prevent instruction pipeline from operating at its maximum rate:

1. **Different segments may take different times** to operate on the incoming information.
2. **Some segments may need to be skipped** for certain operations (for instance, a register mode instruction does not need an effective address computation phase).
3. **Two or more segments may require memory access at the same time**, causing one segment to wait until the other is finished.

What happens in an instruction pipeline

when a branch instruction is encountered?

Design of an Instruction Pipeline: Example

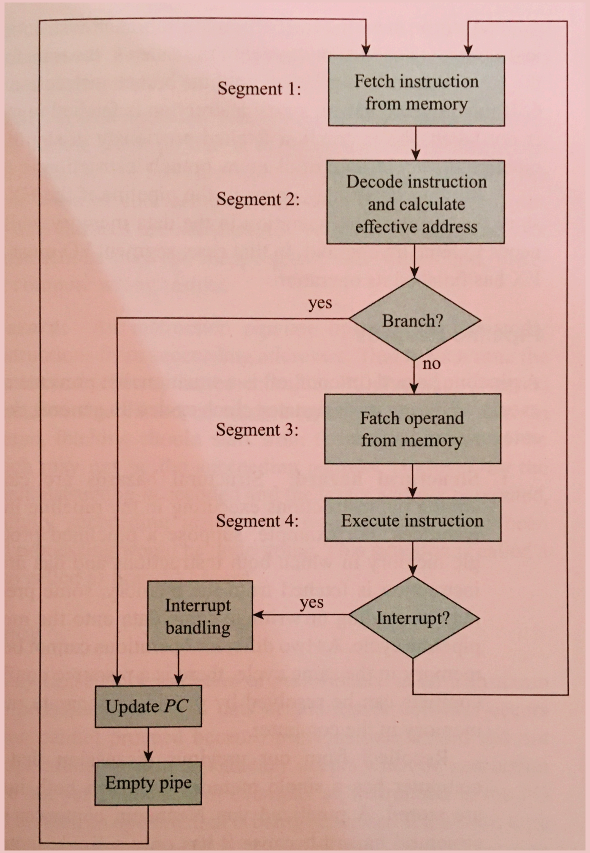
Let’s see how a 4-stage instruction pipeline can be designed.

Assumption 1: *Instruction decoding* phase can be combined with *effective address computation* phase into one segment.

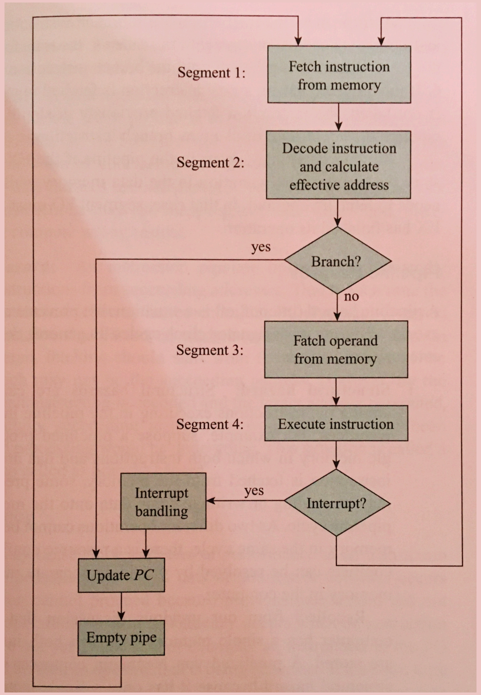
Assumption 2: Most of the instructions place their results in a processor register so that the *instruction execution* phase and the *result storage* phase can be combined into one segment.

Design of an Instruction Pipeline:

4-Segment CPU Pipeline



4-Segment CPU Pipeline

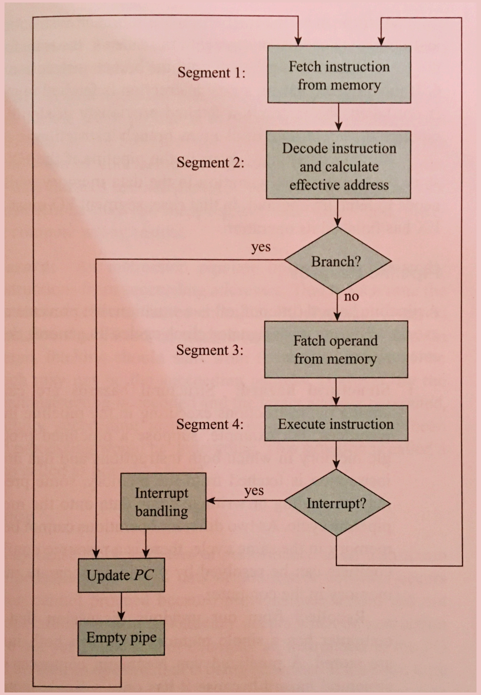


While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching operands from memory in segment 3.

The EA can be computed in a separate circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in the instruction FIFO buffer.

Thus, up to four sub-operations in the instruction cycle can overlap and up to four different instructions can be processed at the same time.

4-Segment CPU Pipeline



Every once in a while, an instruction may cause a branch out of normal sequence. In that case, the pending operations in the last two segments are completed and all information stored in the buffer is deleted.

The pipeline then restarts from the new address stored in PC.

Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address.

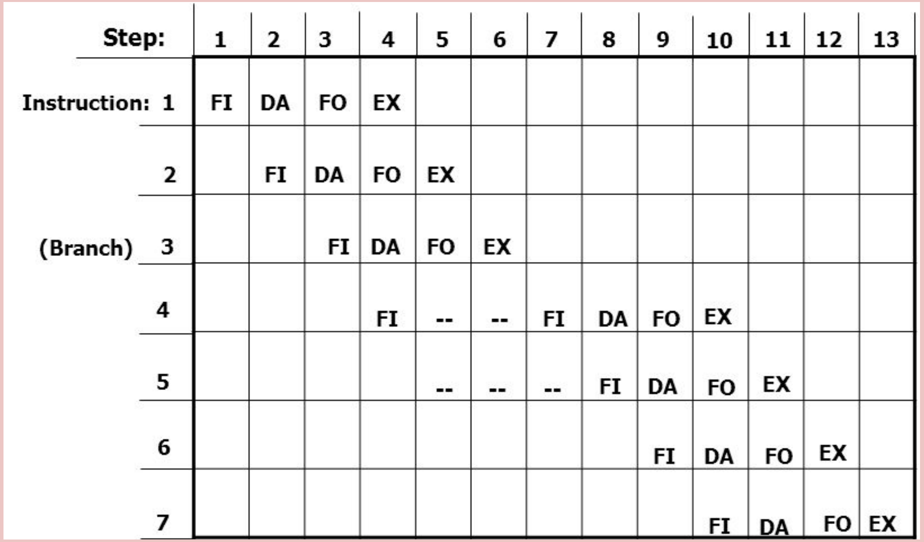
Operations of Instruction Pipeline: Timing Diagram

FI: Segment that fetches instruction

DA: Segment that decodes instruction and computes EA

FO: Segment that fetches operand

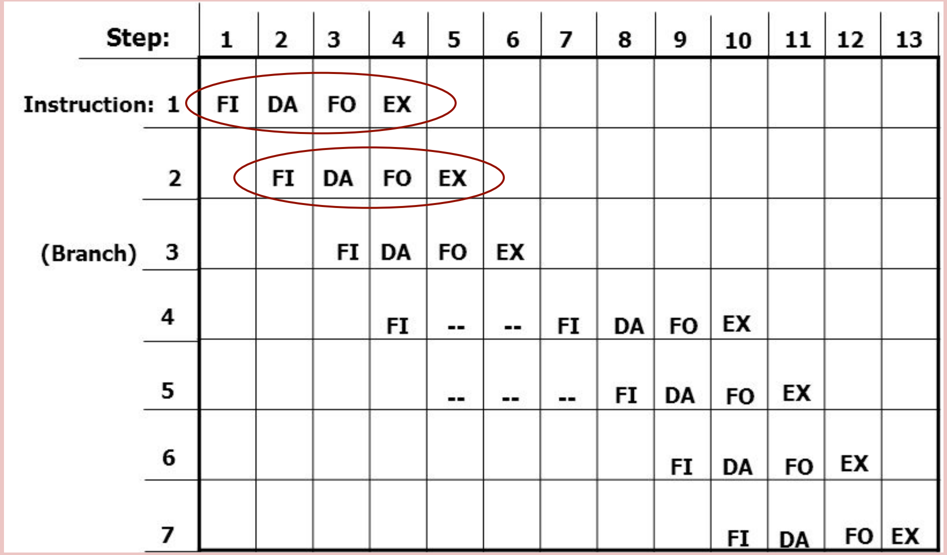
EX: Segment that executes instruction



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

Assumption: Processor has separate instruction and data memories, so that FI and FO can proceed at the same time.

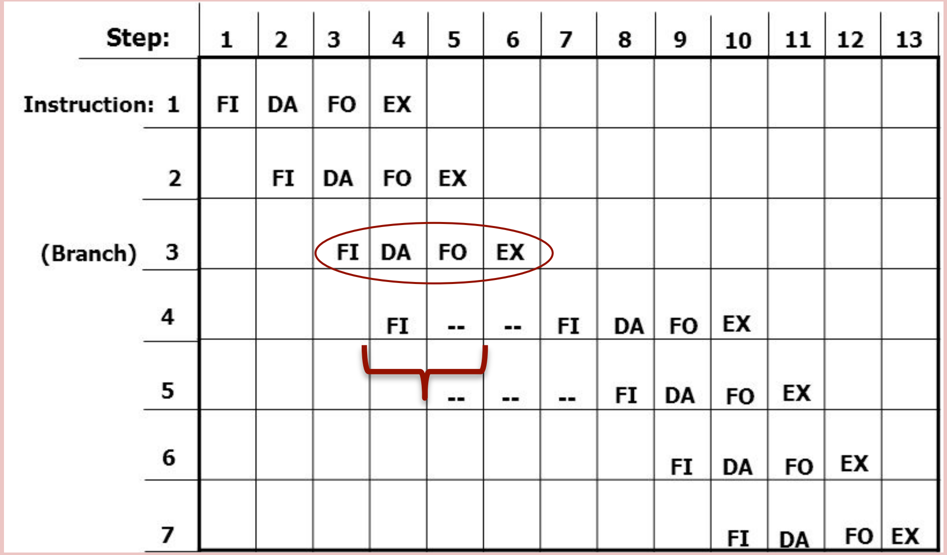
Operations of Instruction Pipeline



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

In the absence of a branch instruction, each segment operates on different instructions.

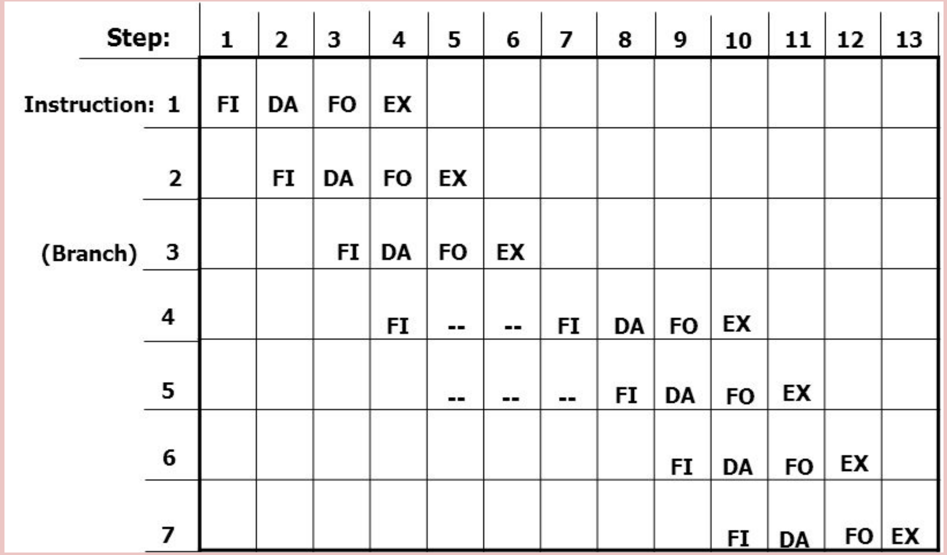
Operations of Instruction Pipeline



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

When a branch instruction (instruction 3) is decoded in step 4, transfer from FI to DA of the other instructions is halted until instruction 3 is executed in step 6.

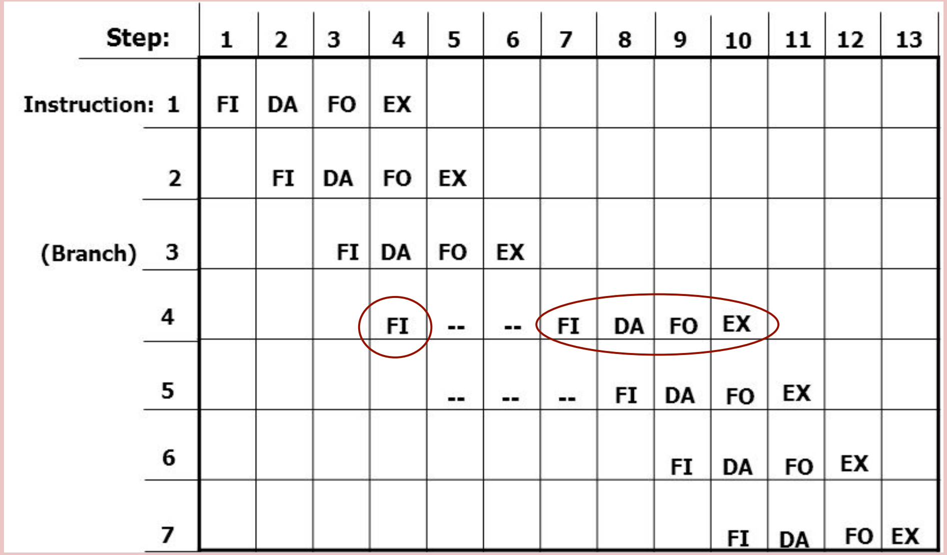
Operations of Instruction Pipeline



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

If the branch is taken, a new instruction is fetched in step 7.

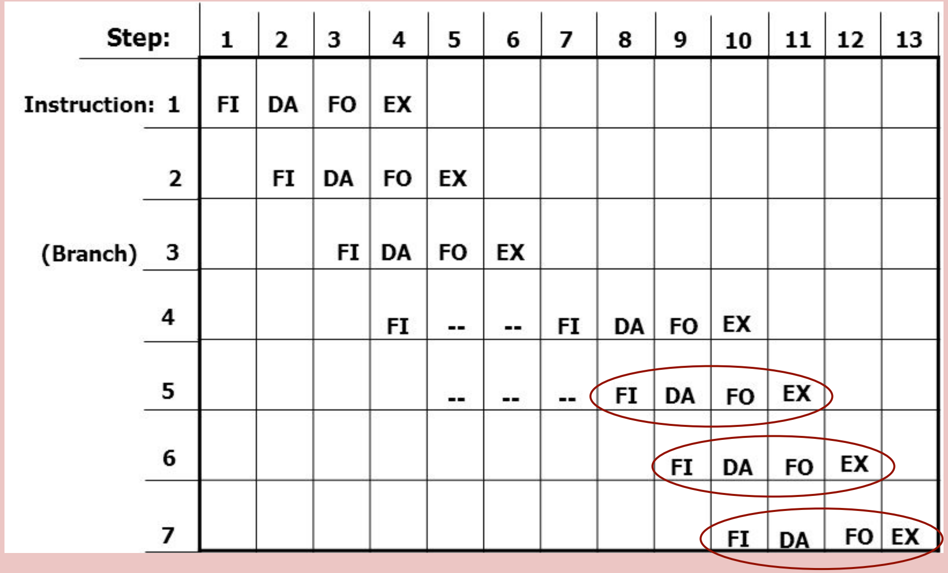
Operations of Instruction Pipeline



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

If the branch is not taken,
the instruction fetched in step 4
can be used

Operations of Instruction 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

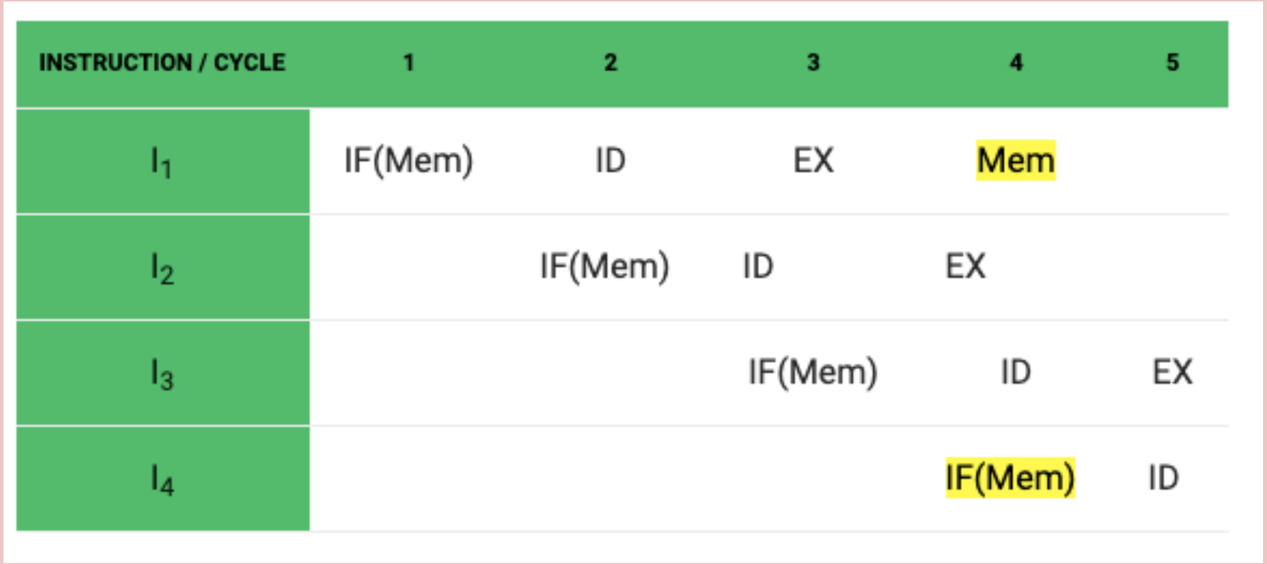
The pipeline then continues until a new branch instruction is encountered.

Pipeline Hazards

Difficulties that cause the instruction pipeline to deviate from its normal operation:

1. **Resource Conflicts:** Caused by accesses to memory by two segments at the same time (aka STRUCTURAL HAZARDS).
2. **Data Dependency Conflicts**: Arise when an instruction depends on the result of previous instruction (aka DATA HAZARDS).
3. **Branch Difficulties:** May arise from branch and other instructions that change the value of PC (aka CONTROL HAZARDS).

Pipeline Hazards: Structural Hazard



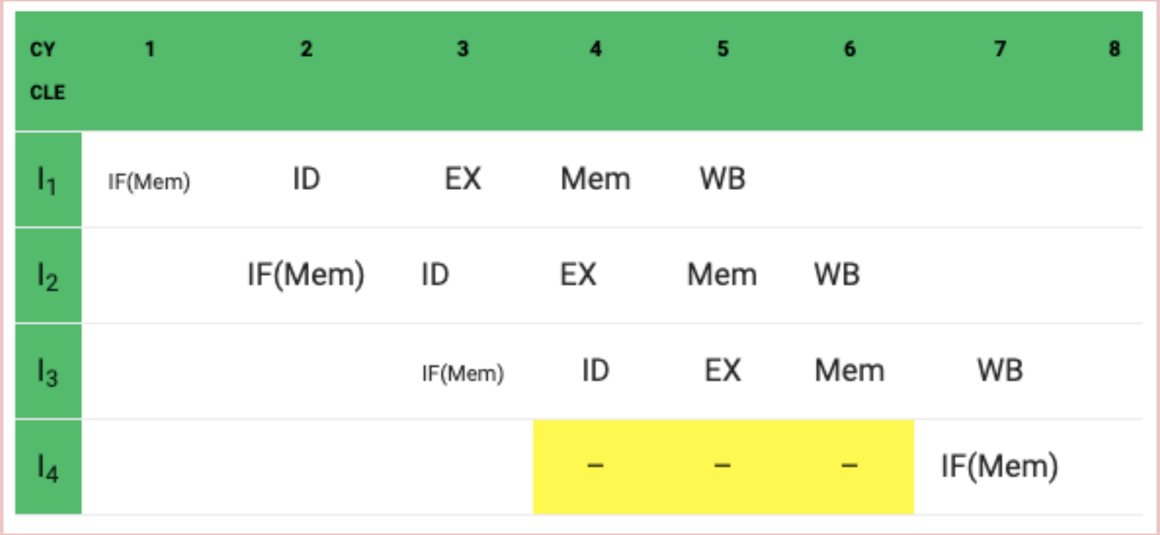
INSTRUCTION / CYCLE	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

In the above scenario, in cycle 4, instructions I_1 and I_4 are trying to access same resource (Memory), which introduces a resource conflict.

Pipeline Hazards: Structural Hazard

To avoid this problem, we have to keep I_4 waiting until memory becomes available.

This wait will introduce ***stalls*** in the pipeline as shown below:



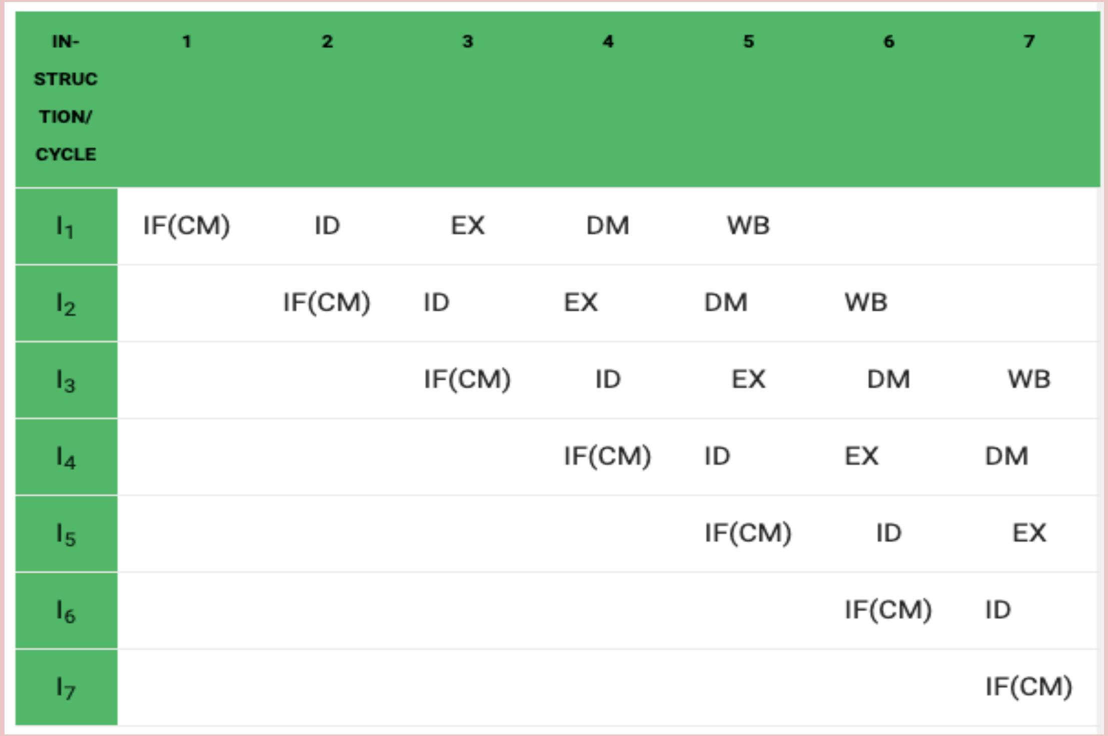
CYCLE	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				–	–	–	IF(Mem)	

Structural Hazard: Solution

To minimize structural dependency stalls in a pipeline, a hardware mechanism called *renaming* is used.

Renaming: Memory is divided into two independent modules Code Memory (CM) and Data Memory (DM) to store the instruction and data separately.

Structural Hazard: Renaming



IN-STRUCTION/CYCLE	1	2	3	4	5	6	7
I_1	IF(CM)	ID	EX	DM	WB		
I_2		IF(CM)	ID	EX	DM	WB	
I_3			IF(CM)	ID	EX	DM	WB
I_4				IF(CM)	ID	EX	DM
I_5					IF(CM)	ID	EX
I_6						IF(CM)	ID
I_7							IF(CM)

Pipeline Hazards: Data Dependency

Data dependency occurs when an instruction needs data (including addresses) that are not yet available.

For instance, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by a previous instruction in segment EX.

Or, an instruction with register indirect mode cannot fetch an operand if the previous instruction is loading the address into register.

Data Dependency: Example

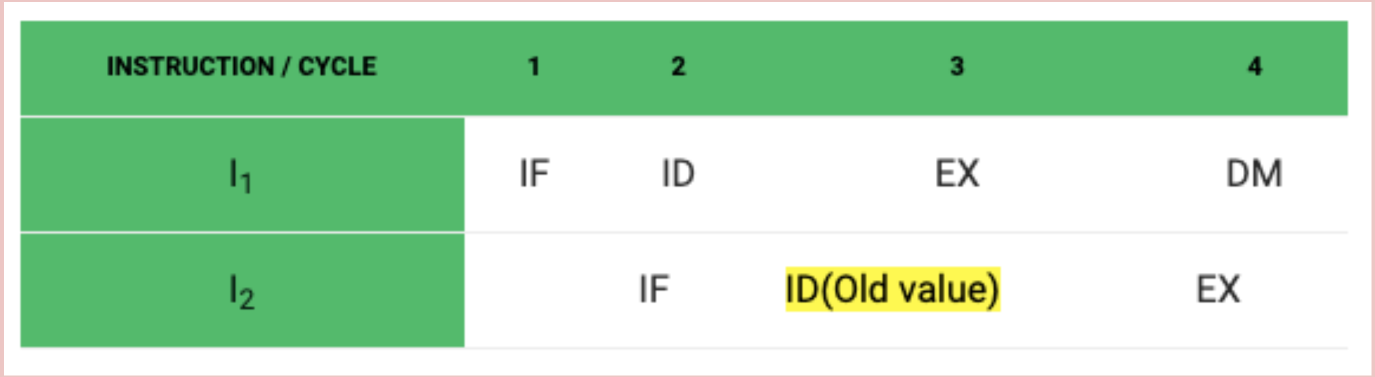
Let there be two instructions I_1 and I_2 such that:

I_1 : ADD R1, R2, R3

I_2 : SUB R4, R1, R2

When these instructions are executed in a pipelined processor, data dependency condition occurs, which means that I_2 tries to read the data before I_1 writes it. Eventually, I_2 incorrectly gets an old value from I_1.

Data Dependency: Example



INSTRUCTION / CYCLE	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID(Old value)	EX

Data Hazards

Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. There are mainly three types of data hazards:

1. **RAW (Read After Write) [Flow/True Data Dependency]**
2. **WAR (Write After Read) [Anti-Data Dependency]**
3. **WAW (Write After Write) [Output Data Dependency]**

Data Hazards: RAW

For two instructions I and J (where J follows I), RAW hazard occurs when instruction J tries to read data before instruction I writes it.

For instance:

I: R2 <- R1 + R3

J: R4 <- R2 + R3

Data Hazards: WAR

For two instructions I and J (where J follows I), WAR hazard occurs when instruction J tries to write data before instruction I reads it.

For instance:

I: R2 <- R1 + R3

J: R3 <- R4 + R5

Data Hazards: WAW

For two instructions I and J (where J follows I), WAW hazard occurs when instruction J tries to write output before instruction I writes it.

For instance:

I: R2 <- R1 + R3

J: R2 <- R4 + R5

Dealing With Data Dependency Conflicts

>> Hardware Interlocks

>> Operand Forwarding

>> Delayed Load

Dealing With Data Dependency Conflicts: **Hardware Interlocks**

>> Most straightforward method.

>> Entails insertion of hardware interlocks, which are circuits that detect instructions whose source operands are destinations of instructions farther up in the pipeline.

>> Upon detection of such situations, instruction whose source is not yet available is delayed by enough clock cycles to resolve the conflict.

Dealing With Data Dependency Conflicts: **Hardware Interlocks**

Consider the following sequence of instructions in a program:

I_1 : ADD R1, R2, R3

…

I_3 : SUB R6, R4, R5

I_4 : MUL R7, R1, R8

R1 is a source operand for I_4; it is also a destination operand for I_1.

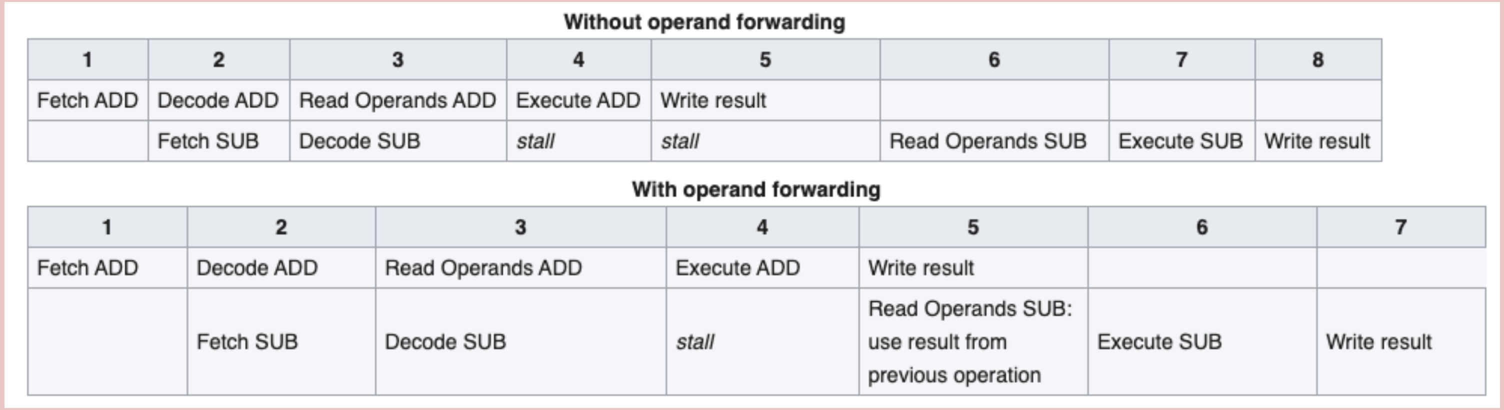
Dealing With Data Dependency Conflicts: **Operand Forwarding**

>> Entails use of special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.

>> Interface registers are placed between segments to hold intermediate output; dependent instruction can access new value from the interface register directly.

Operand Forwarding: Example

ADD R3, R2, R0
SUB R4, R3, 8



Without operand forwarding

1	2	3	4	5	6	7	8
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result			
	Fetch SUB	Decode SUB	*stall*	*stall*	Read Operands SUB	Execute SUB	Write result

With operand forwarding

1	2	3	4	5	6	7
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result		
	Fetch SUB	Decode SUB	*stall*	Read Operands SUB: use result from previous operation	Execute SUB	Write result

Dealing With Data Dependency Conflicts: **Delayed Load**

>> Responsibility for resolving conflicts is given to the compiler that translates high-level program into a machine language program.

>> Compiler is designed to detect conflicts and reorder the instructions to **delay the loading of conflicting data** by inserting no-operation instructions.

Pipeline Hazards: Branch Dependency aka Control Hazard

Arise from instructions like BRANCH, CALL, or JMP that change the value of PC.

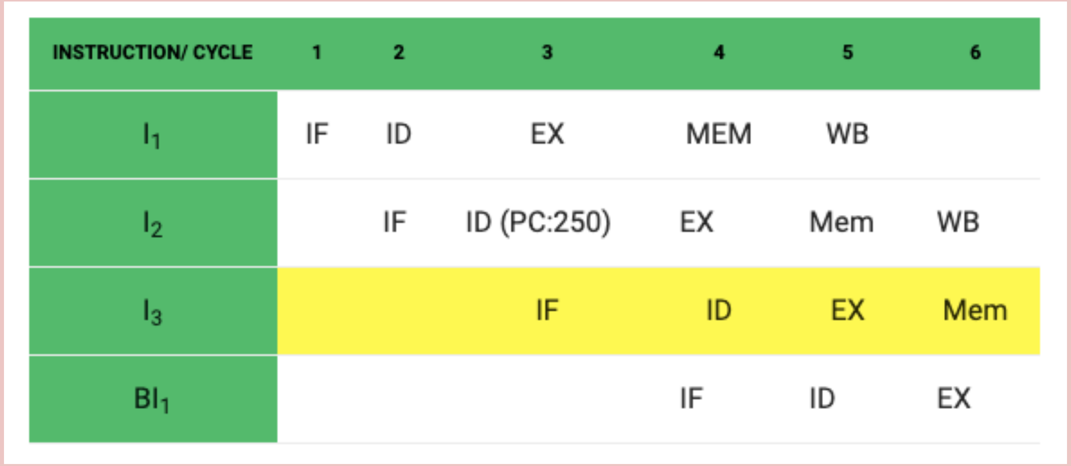
Consider the following sequence of instructions in a program:

100: I_1
101: I_2 (JMP 250)
102: I_3
.
.
250: BI_1

Expected output: I_1 -> I_2 -> BI_1

Branch Dependency aka Control Hazard: Example

Expected output: I_1 -> I_2 -> BI_1



INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

Output Sequence: I_1 -> I_2 -> I_3 -> BI_1

Handling Branch Instructions

>> Prefetch Target Instruction

>> Branch Target Buffer

>> Loop Buffer

>> Branch Prediction

>> Delayed Branch

Handling Branch Instructions: Prefetch Target Instructions

>> **Prefetch the target instruction in addition to the instruction following the branch**.

Both are saved until the branch is executed.

>> If the branch condition is successful, the pipeline continues from the branch target instruction.

Handling Branch Instructions: Branch Target Buffer (BTB)

>> BTB, an associative memory, is included in the fetch segment of the pipeline.

>> **Each entry of the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction.**

>> When the pipeline decodes a branch instruction, it also searches BTB for the address of the instruction. If it is found in BTB, the instruction is available directly. If not, the pipeline shifts to a new instruction stream (and stores target instruction in BTB).

Handling Branch Instructions: Loop Buffer

>> A variation of BTB; it is a small, very high speed register file maintained by the IF segment.

>> **When a program loop is detected, it is stored in the loop buffer in its entirety, including all branches.**

>> The program loop can be executed directly without having to access memory until the loop mode is removed from the final branching out.

Handling Branch Instructions: Branch Prediction

>> A pipeline with branch prediction **uses some additional logic to guess the outcome of a conditional branch instruction before it is executed**.

>> The pipeline then begins **prefetching the instruction stream from the predicted path**.

>> A correct prediction eliminates the wasted time caused by *branch penalties*.

Handling Branch Instructions: Delayed Branch

>> Procedure employed in most RISC processors.

>> Compiler **detects the branch instructions and rearranges the code sequence** by inserting useful instructions that keep the pipeline operating without interruptions.

Multiprocessors

Multiprocessor Systems

A multiprocessor system is an interconnection of **two or more CPUs** with **memory** and **input-output equipment**. They are MIMD systems.

The term “processor” in *multiprocessor* can mean either a CPU or an IOP.

A system with a single CPU and and one or more IOPs is usually not included in the definition of a multiprocessor system, unless the IOPs have computational facilities comparable to a CPU.

Multiprocessor Systems Versus Multicomputer Systems

System with Multiple Computers

Computers are interconnected with each other by means of communication lines to form a *computer network*.

The network consists of several autonomous computers that **may or may not communicate with one another.**

System with Multiple Processors

Such a system is controlled by one OS that provides interaction between processors, and all the components of the system cooperate in the solution of a problem.

Advantages of Multiprocessor Systems

Multiprocessing improves reliability of the system

so that a failure or error in one part has a limited effect on the rest of the system.

Multiprocessing results in reduced loss of efficiency

in case of failure of a processor, because if one processor fails,

another can be assigned to perform its functions.

Advantages of Multiprocessor Systems (continued)

Multiprocessing improves system performance,
primarily because computations can proceed in parallel in one of two ways:

1. Multiple independent jobs can be made to operate in parallel.

2. A single job can be partitioned into multiple parallel tasks.

Classification of Multiprocessors

Tightly Coupled aka Shared-Memory Multiprocessors

Multiprocessor systems with common shared memory.

The CPUs still have their own local memories as well.

Loosely Coupled aka Distributed-Memory Multiprocessors

Each processor has its own private local memory.
The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme.

Interconnection Structures

Interconnection Structures

Components that form a multiprocessor system are CPUs, IOPs, and a memory unit that may be partitioned into a number of separate modules. Interconnections between these components can have different configurations, the most common amongst which are:

Time-Shared Common Bus

Multiport Memory

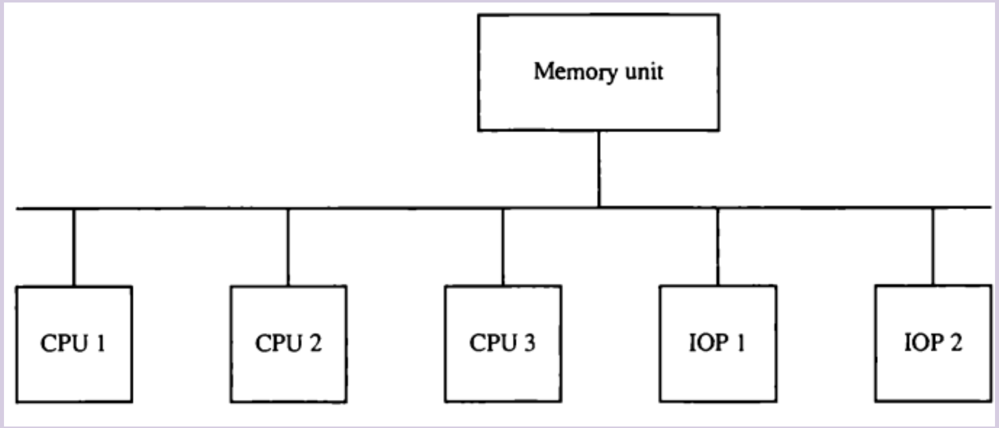
Crossbar Switch

Multistage Switching Network

Hypercube System

Interconnection Structures: Time-Shared Common Bus

A number of processors are connected to the memory unit through a common path.



Only one processor can communicate with the memory or another processors at any given time.

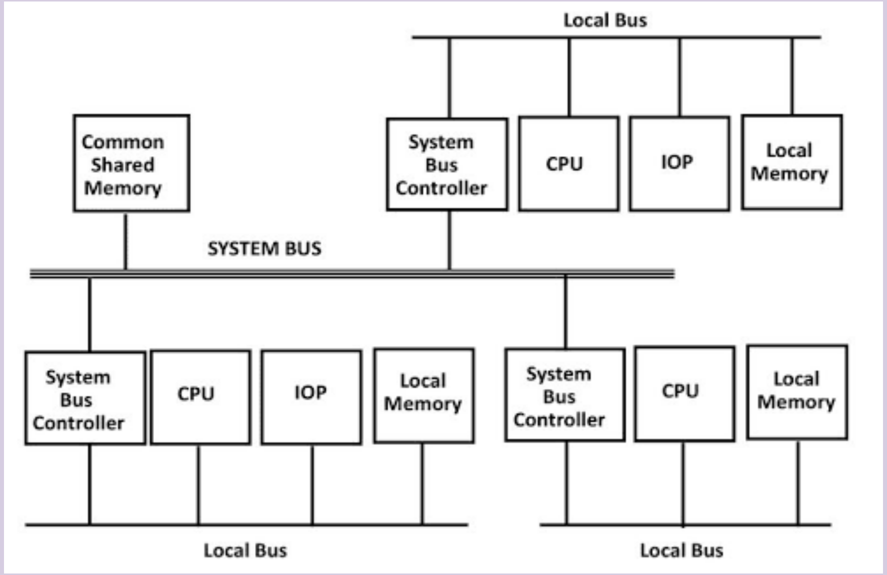
Limitation of Time-Shared Common Bus

>> enables one transfer at a time

>> the overall transfer rate is limited by the speed of the single path

The processors in the system can be kept busy more often through the implementation of ***two or more independent buses*** *to permit multiple simultaneous bus transfers.*

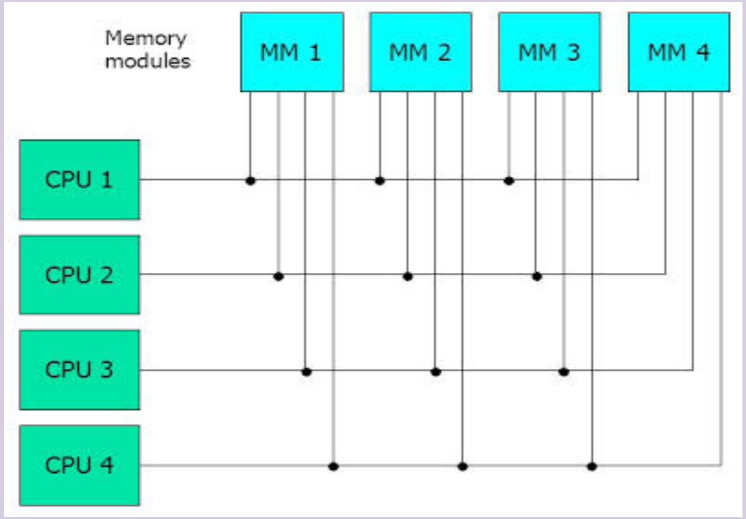
Dual Bus Structure



System bus structure for multiprocessors

Interconnection Structures: Multiport Memory

A multiport memory system employs separate buses between each memory module and CPU.



Memory access conflicts are resolved by assigning fixed priorities to each memory port.

Characteristics of Multiport Memory Organization

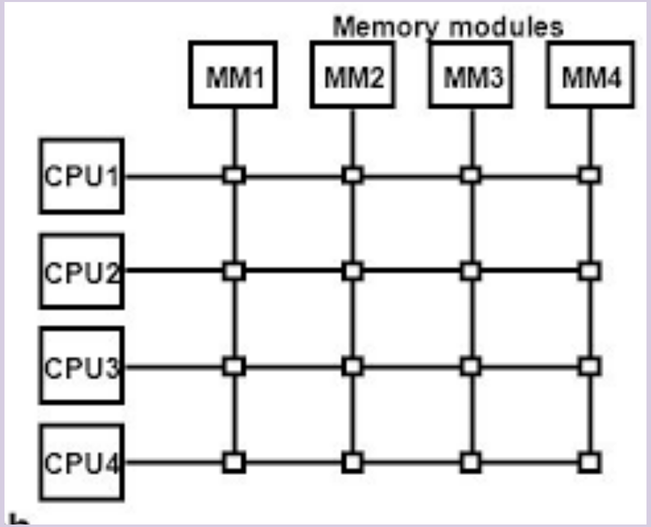
ADVANTAGE: High transfer rate

DISADVANTAGE: Requires expensive memory control logic and a large number of cables and connectors.

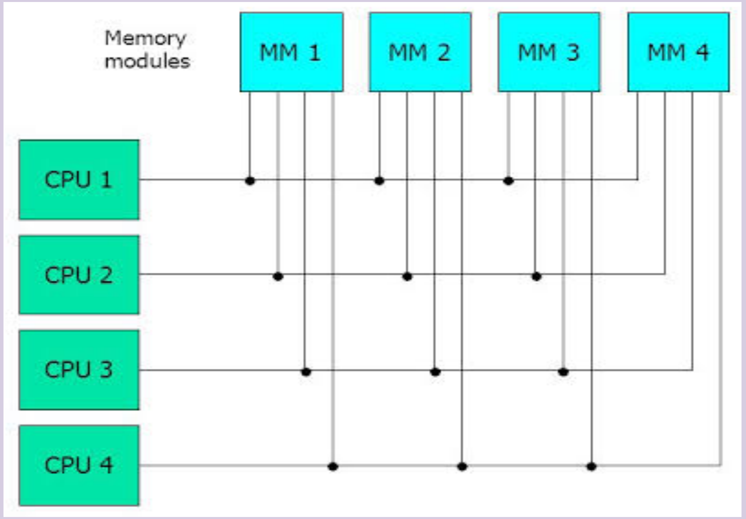
Mainly used in systems with a small number of processors.

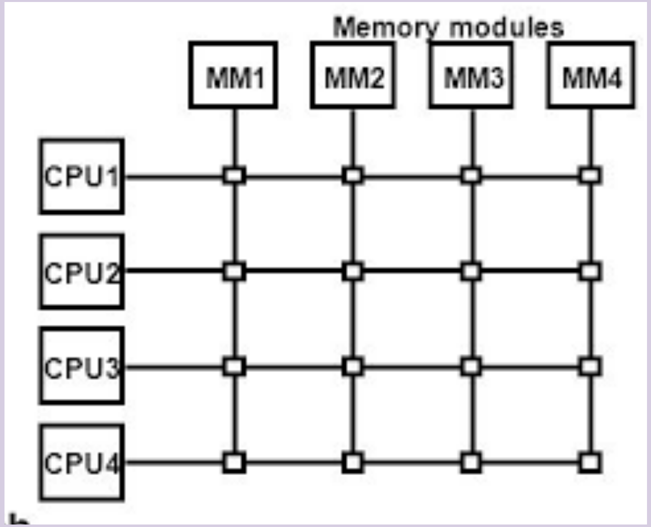
Interconnection Structures: Crossbar Switch

This system consists of a number of crosspoints placed at intersections between processor buses and memory module paths

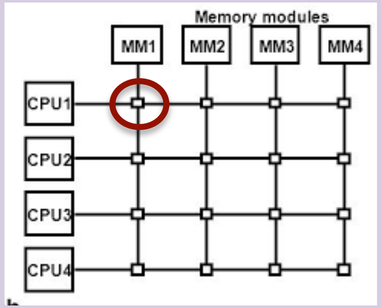


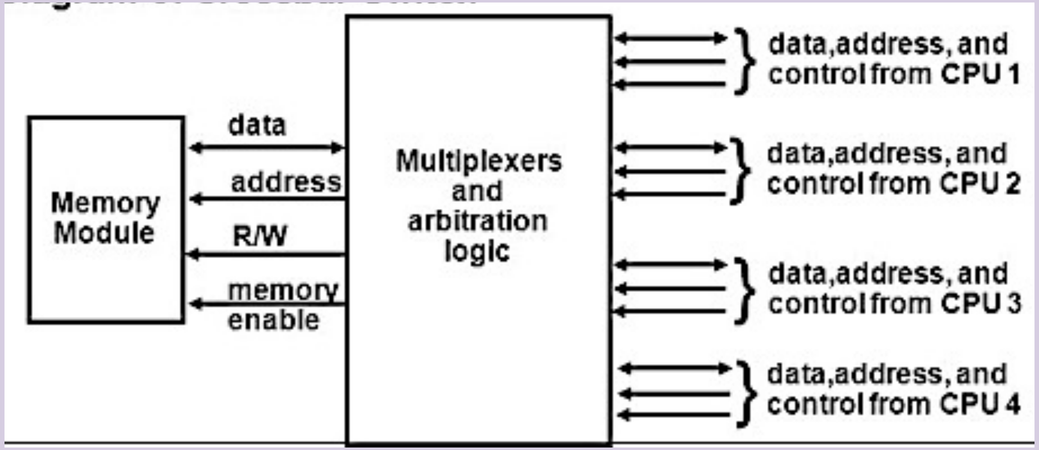
Multiport Memory vs Crossbar Switch





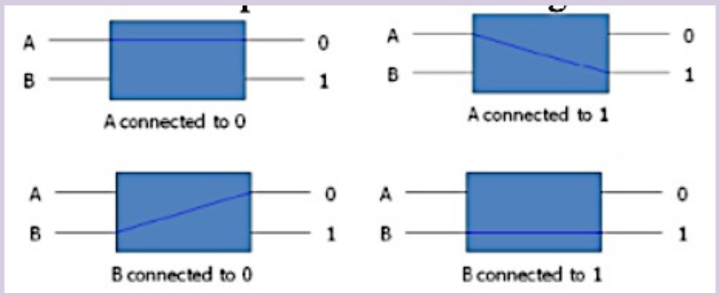
Crossbar Switch




Memory Module
data
address
R/W
memory enable
Multiplexers and arbitration logic
data,address, and control from CPU 1
data,address, and control from CPU 2
data,address, and control from CPU 3
data,address, and control from CPU 4

Interconnection Structures: Multistage Switching Network

The main component of a multistage switching network is a two-input, two-output interchange switch.

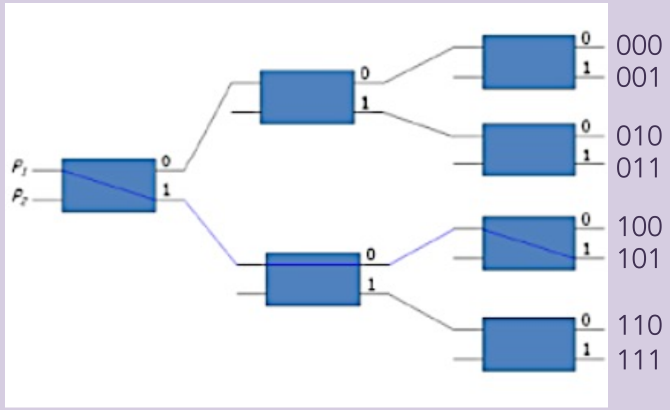


Operation of a 2 x 2 interchange switch

Interconnection Structures: Multistage Switching Network

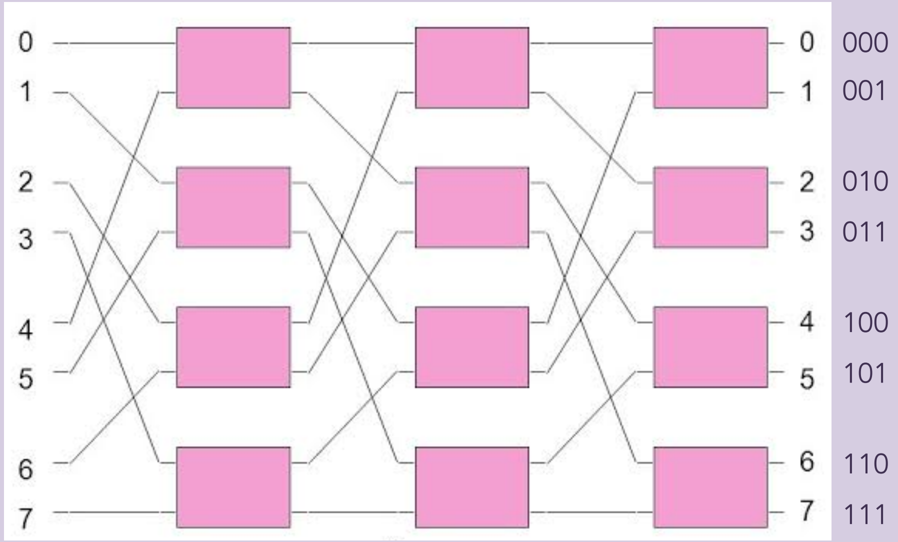
Using the 2 x 2 switch, it is possible to build a multistage network to control the communication between a number of sources and destinations.

Consider for instance, the connection between two processes and eight memory modules:



Binary tree with 2 x 2 switches

Interconnection Structures: Multistage Switching Network

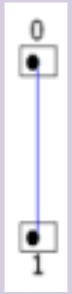


8 x 8 omega switching network

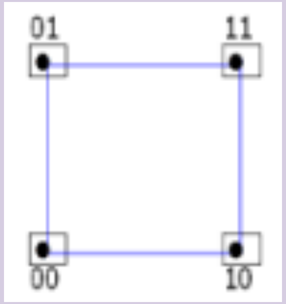
Interconnection Structures: Hypercube System

A hypercube, or a binary *n*-cube multiprocessor structure, is a loosely coupled system composed of N = 2^n processors interconnected in a *n*-dimensional binary cube.

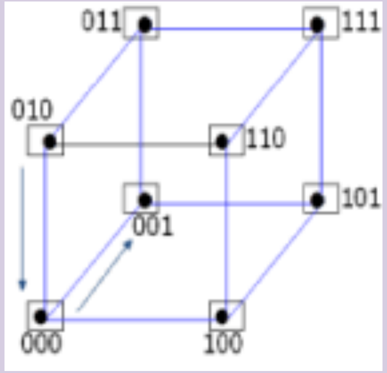
Each processor forms a *node* of the cube.



One-Cube



Two-Cube



Three-Cube

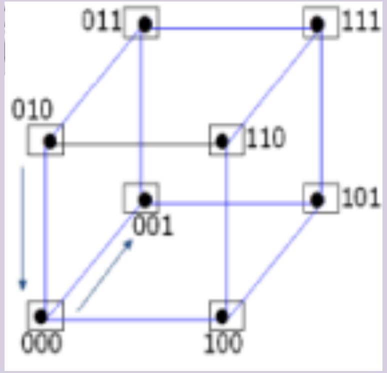
Hypercube Interconnection: Routing Procedure

To route messages through an n-cube structure, take the XOR of the source node address and the destination node address.

The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes.

Say, a message needs to go from 010 to 001. The XOR is 011 (the second and third bits are 1).

So, the message can be sent from 010 to 000 (two nodes whose addresses differ in the second bit) and then from 000 to 001 (two nodes whose addresses differ in the third bit).



Interprocessor Arbitration

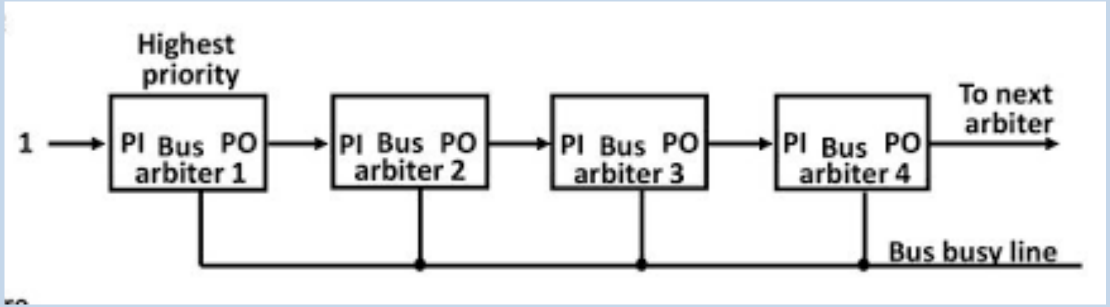
Bus Arbitration

Processors in a shared-memory multiprocessor system request access to common resources through the system bus. If no other processor is currently using the bus, the requesting processor may be granted access immediately.
Otherwise, the requesting processor must wait.

Arbitration must then be performed to resolve this multiple contention for the shared resources.
This logic must be placed between the local bus and the system bus.

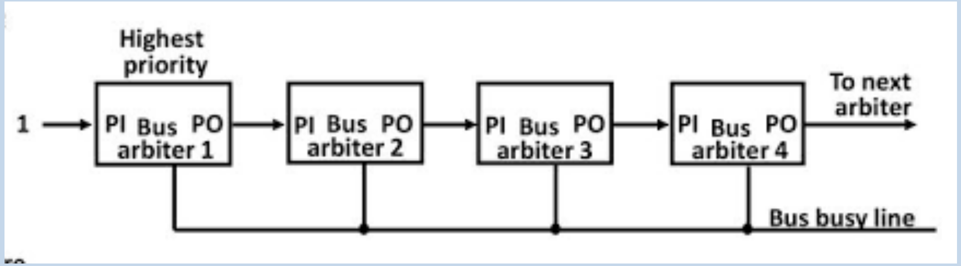
Bus Arbitration: Serial Arbitration Procedure

Arbitration procedures service all requests on the basis of established priorities.
A serial priority resolving technique can be established using a daisy-chain connection of bus arbitration circuits.



In this method, processors connected to the system bus are assigned priority according to their position along the priority control line.

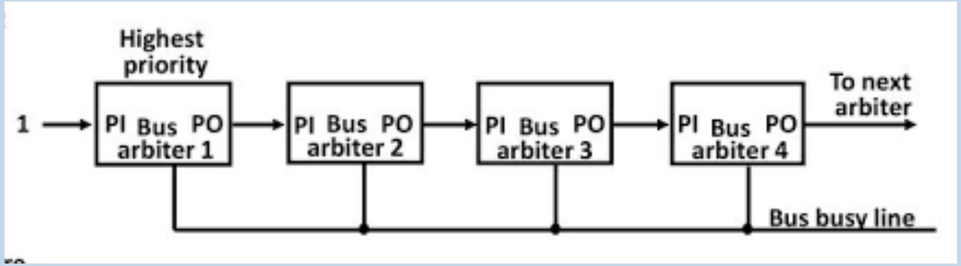
Serial Daisy-Chain Arbitration



PI of the highest-priority unit is always 1 (which implies that the highest-priority unit will always receive access to the system bus when it requests it).

PO of a particular arbiter is 1 if its PI is 1 and the processor associated with this arbiter is not requesting control of the bus (this way, priority is passed to the next unit in the chain).

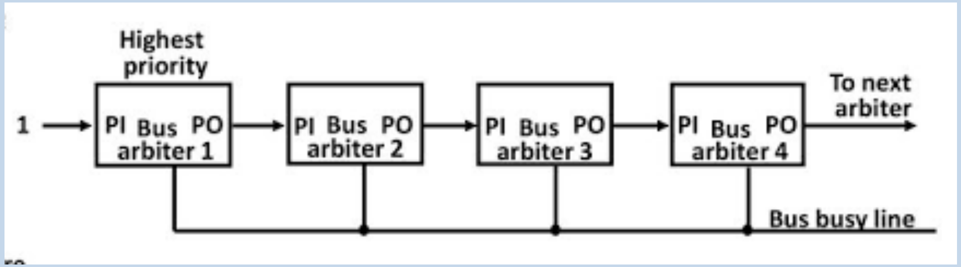
Serial Daisy-Chain Arbitration



If a processor requests control of the bus and its arbiter finds that PI = 1, it sets its PO = 0 (this way, control of the bus is restricted from passing to the next unit).

When a lower-priority arbiter finds its PI = 0, it sets its PO = 0.

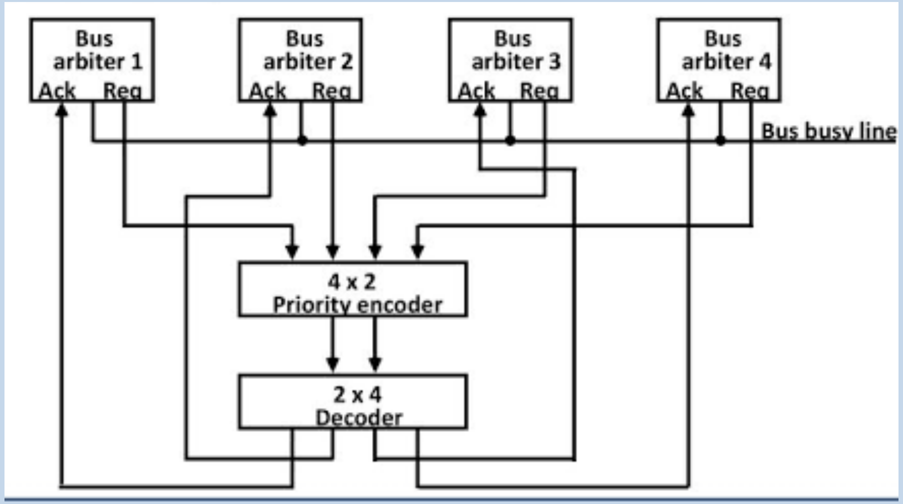
Serial Daisy-Chain Arbitration



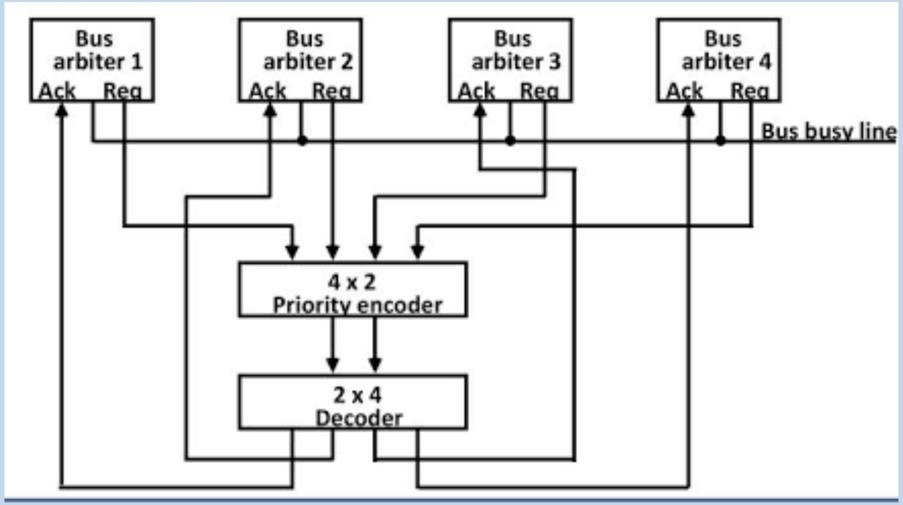
An arbiter whose PI = ? and PO = ? is the one who is given control of the bus.

Bus Arbitration: Parallel Arbitration Logic

A parallel bus arbitration technique uses an external priority encoder and a decoder.



Parallel Arbitration Logic



Each bus arbiter has a **bus request** output line and a **bus acknowledge** input line. *How are they used?*

Priority Encoder: Generates a 2-bit code that represents the highest priority unit amongst those requesting the bus.

Priority Decoder: Receives the 2-bit code, which drives it to enable the proper acknowledge line to grant bus access to the highest-priority unit.

Bus Arbitration: Dynamic Algorithms

The two arbitration techniques described earlier use a **static priority algorithm** (since the priority of each device is fixed by the way it is connected to the bus).

Dynamic priority algorithms allow the priority of the devices to be changed while the system is in operation.

Algorithms:

Time-Slice ***Polling*** ***LRU*** ***FCFS*** ***Rotating Daisy-Chain***

Dynamic Bus Arbitration Algorithms

Time-Slice: Fixed-length time slices of bus time are offered to each processor sequentially, in a round-robin fashion.

Polling: Bus grant signal is replaced by a set of lines called *poll lines,* which are connected to all units. These lines are used by the bus controller to define an address for each device. The controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the busy bus line and then accesses the bus. After a number of bus cycles, the polling continues to choose the next processor.

Dynamic Bus Arbitration Algorithms (continued)

LRU: Gives the highest priority to the requesting device that has not used the bus in the longest period of time.

FCFS: Requests are served in the order they are received.

Rotating Daisy-Chain: Dynamic extension of daisy-chain algorithm. Here, there is no central bus controller, and the priority line is connected from the PO of the last device to PI of the first in a closed loop. Once an arbiter releases the bus after using it, it has the lowest priority.

Interprocessor Communication and Synchronization

Interprocessor Communication

Various processors in a multiprocessor system must be provided with a facility for communicating with each other.

In **tightly coupled (or shared-memory) systems**, a portion of the memory is set aside, which is accessible to all processors. This memory acts as a mailbox for sending and receiving messages.

In **loosely coupled systems**, there is no shared memory, which is why the processors must communicate via message-passing.

Interprocessor Communication: Conflict Prevention

To prevent conflicting use of shared resources, there must be a procedure for assigning resources to processors.

This is done by the OS, using any of the following organizations:

Master-Slave Configuration
Separate Operating Systems
Distributed Operating Systems

Design of OS for Multiprocessors: Master-Slave Configuration

One processor, designated the master, always executes the OS functions.
The remaining processors, i.e., the slaves, do not perform any OS functions.

If a slave processor needs an OS service, it must request
it by interrupting the master and
waiting till the current program can be interrupted.

Design of OS for Multiprocessors: Separate Operating Systems

Any processor can execute the OS routines it needs.

This organization is more suitable in loosely coupled systems, where every processor may have its own copy of the entire OS.

Design of OS for Multiprocessors: Distributed Operating Systems

OS routines are distributed amongst the available processors.

However, each particular OS function is assigned to only one processor at a time.

This organization is referred to as a **floating OS**
since the routines float from one processor to another.

Interprocessor Synchronization

While communication refers to the exchange of data between different processes, synchronization refers to the special case of where the data being communicated is ***control information***.

Synchronization is needed to enforce the correct sequence of processes and ensure mutually exclusive access to shared writable data.

Interprocessor Synchronization: Binary Semaphores

In a multiprocessor system, it is crucial to ensure that data is not being changed by two processes simultaneously. This mechanism is termed **mutual exclusion.**

Mutual exclusion must be provided to enable one processor to lock-out access to a shared resource by other processors while it is operating in its **critical section**.

A semaphore is binary variable that is used to indicate whether or not a processor is executing its critical section.

Semaphores

A semaphore is software controlled flag that is stored in a memory location that all processors can access.

When the **semaphore = 1**, it means the a processor is executing a critical program (and the shared memory is not available to other processors).

When the **semaphore = 0**, it means the no processor is executing a critical program (and the shared memory is available to other processors).

Semaphore Initialization

A semaphore can be initialized by means of **test-and-set instructions** in conjunction with a **hardware lock** mechanism.

The test-and-set instruction tests and sets a semaphore and activates the lock mechanism during the time that the instruction is being executed.

This prevents other processors from changing the semaphore between the time that the processor is testing it and the time that it is setting it.

Semaphore Initialization: Purpose of the Lock Signal

The lock signal must be active during execution of the test-and-set instruction, but it does not have to be active once the semaphore is set.

Thus, the lock mechanism prevents other processors from accessing memory **while the semaphore is being set**.

The semaphore itself, when set, prevents other processors from accessing shared memory while a processor is executing in its critical section.

Semaphore Initialization

If we use the mnemonic TSL to designate the "test and set while locked" operation and SEM the designate the semaphore's address in memory, then the instruction

```
TSL     SEM
```

will be executed in two memory cycles (one to read and another to write) without interference as follows (R is a processor register):

R ← M[SEM]	**Test Semaphore**
M[SEM] ← 1	**Set Semaphore**

Semaphore Initialization

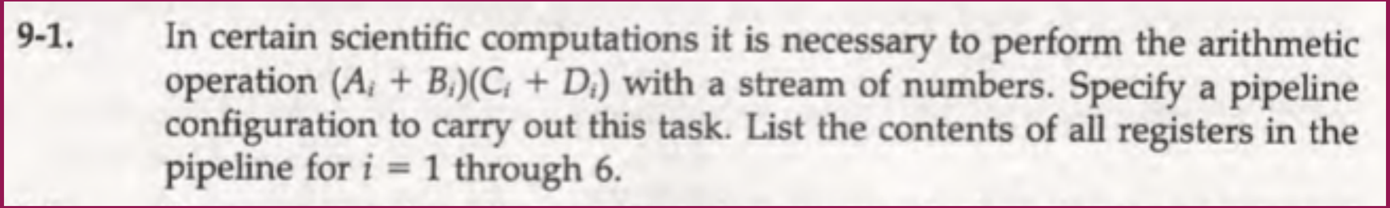
$R \leftarrow M[SEM]$	Test Semaphore
$M[SEM] \leftarrow 1$	Set Semaphore

The value of R determines what to do next.

If a processor finds that $R = 1$, it knows that the semaphore was set by another processor.

If $R = 0$, it means that the common memory (or the shared resource that the semaphore represents) is available.

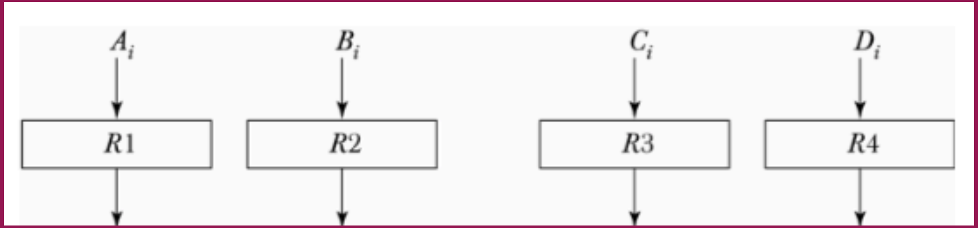
Exercises



9-1. In certain scientific computations it is necessary to perform the arithmetic operation $(A_i + B_i)(C_i + D_i)$ with a stream of numbers. Specify a pipeline configuration to carry out this task. List the contents of all registers in the pipeline for $i = 1$ through 6.

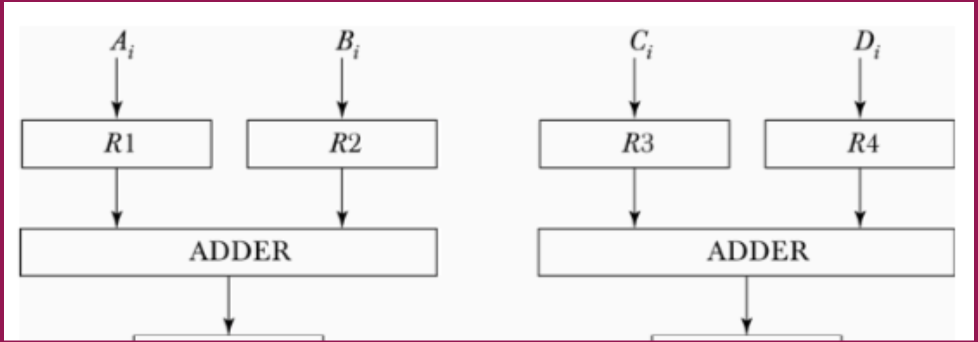
Solution:

$$(A_i + B_i)(C_i + D_i)$$



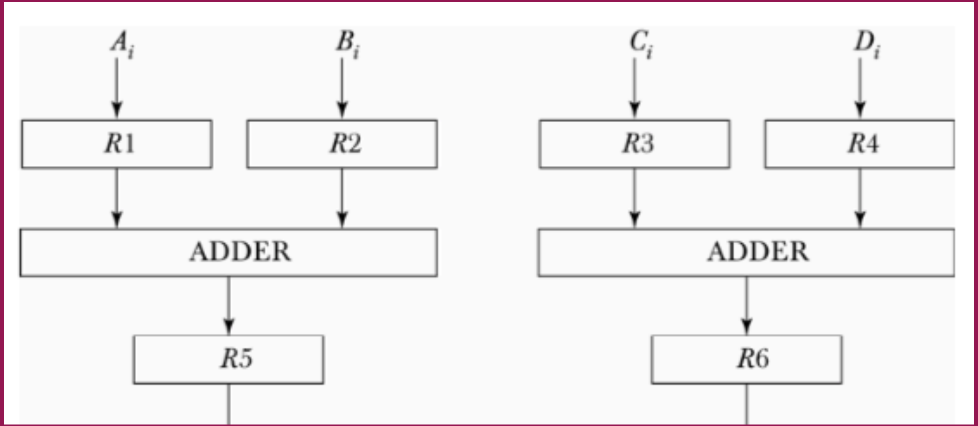
Solution:

$$(A_i + B_i)(C_i + D_i)$$



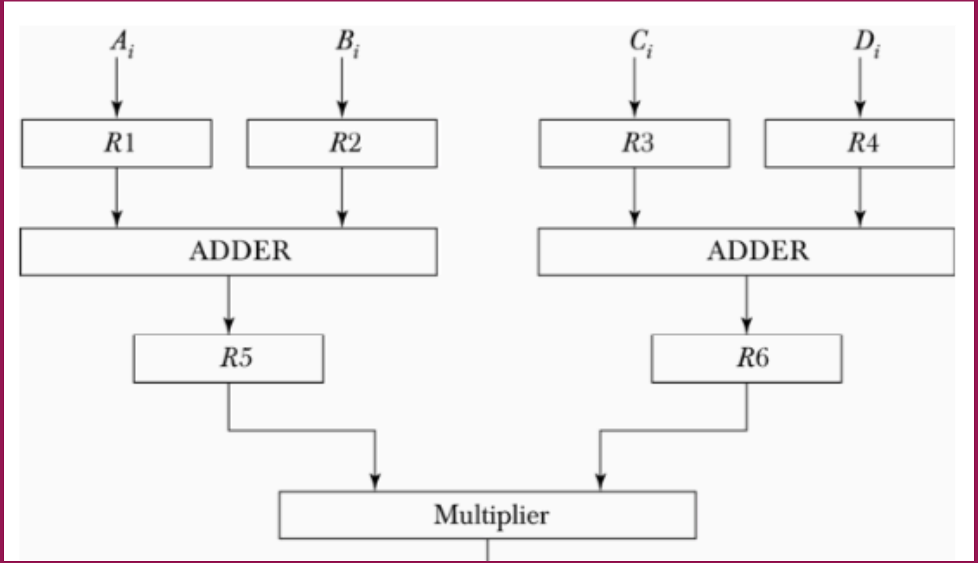
Solution:

$$(A_i + B_i)(C_i + D_i)$$



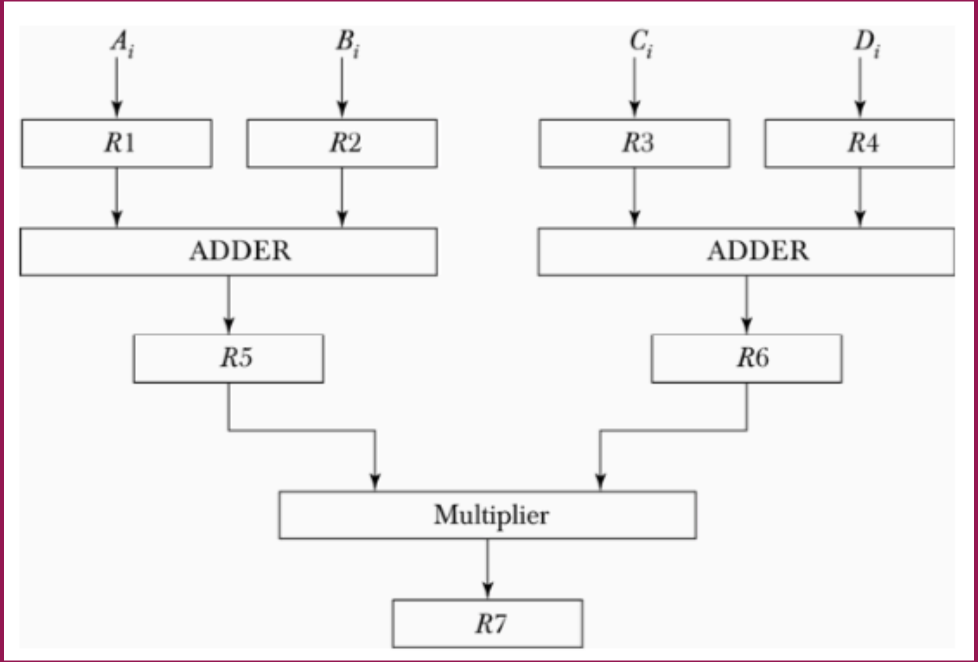
Solution:

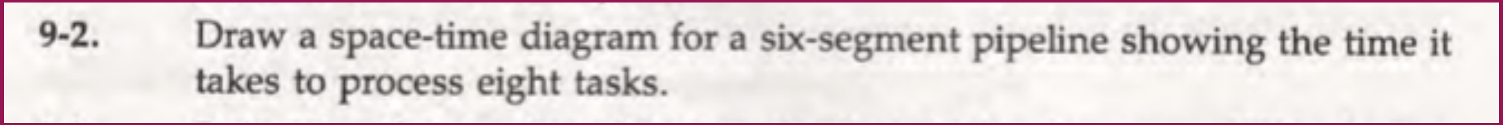
$$(A_i + B_i) (C_i + D_i)$$



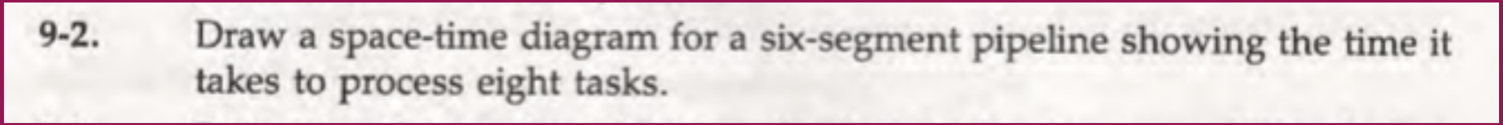
Solution:

$$(A_i + B_i)(C_i + D_i)$$



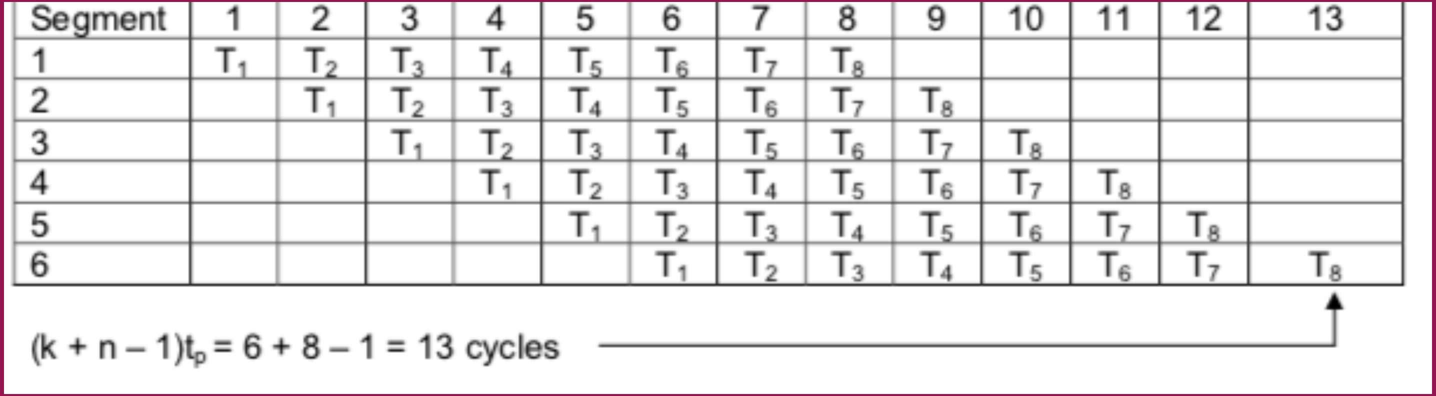


9-2. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.



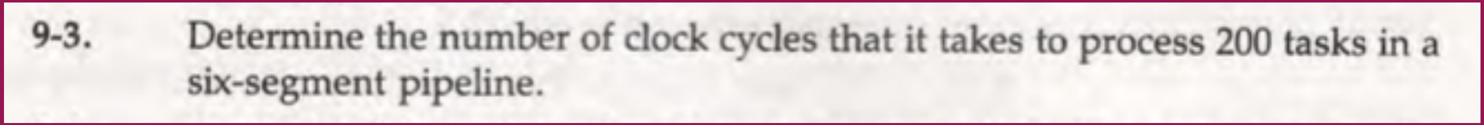
9-2. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

Solution:

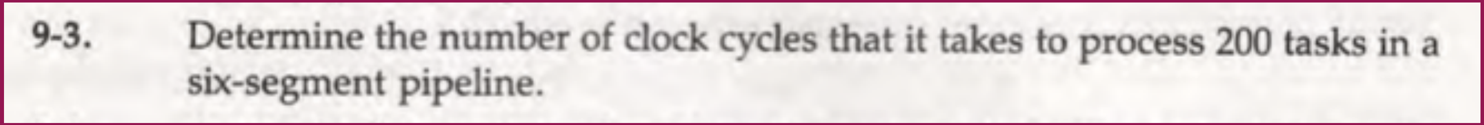


Segment	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8					
2		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8				
3			T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8			
4				T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8		
5					T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
6						T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8

$(k + n - 1)t_p = 6 + 8 - 1 = 13$ cycles



9-3. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.



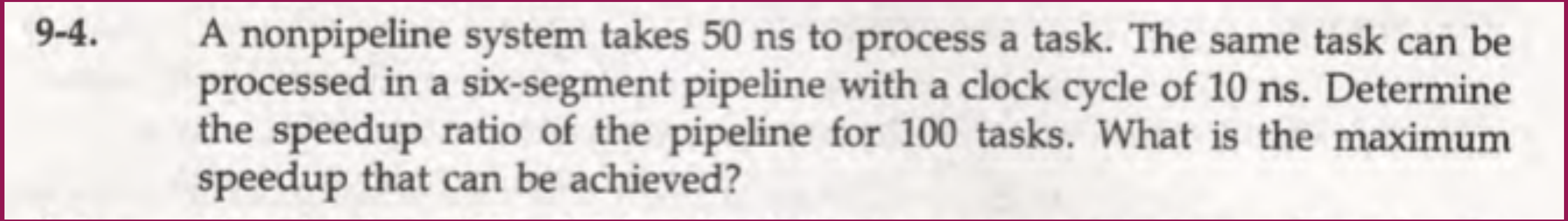
9-3. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.

Solution:

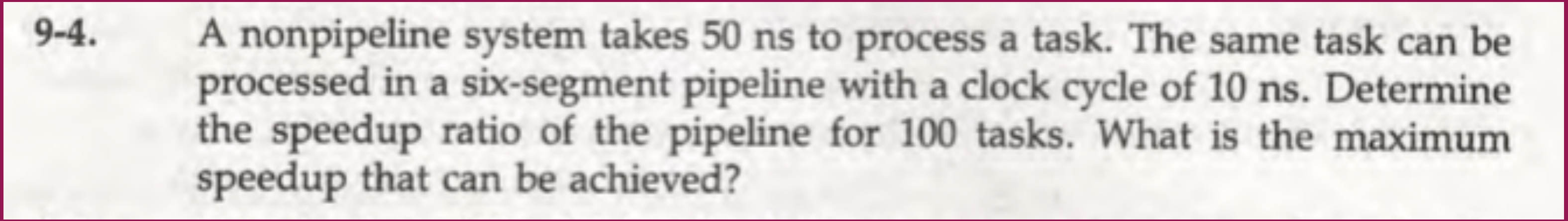
Number of segments (k) = 6

Number of tasks (n) = 200

Clocks cycles required = k + n – 1 = 6 + 200 – 1 = 205



9-4. A nonpipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?


9-4. A nonpipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

Solution:

$t_n = 50$ ns

$k = 6$

$t_p = 10$ ns

$n = 100$

$S = n\, t_n / (k + n - 1)\, t_p = (100 \times 50) / [(6 + 100 - 1) \times 100] = 4.76$

$S_{max} = t_n / t_p = 50 / 10 = 5$

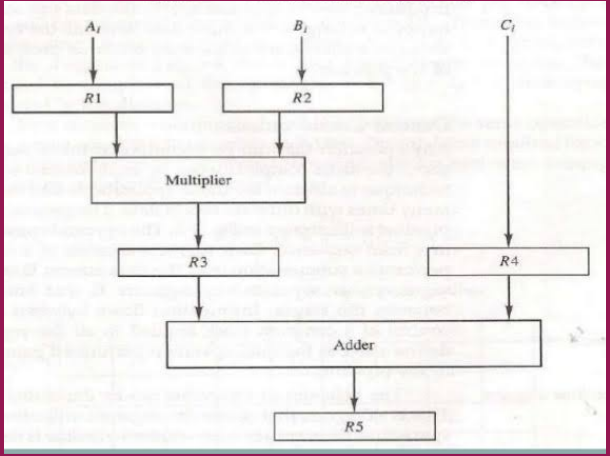
9-5. The pipeline of Fig. 9-2 has the following propagation times: 40 ns for the operands to be read from memory into registers *R*1 and *R*2, 45 ns for the signal to propagate through the multiplier, 5 ns for the transfer into *R*3, and 15 ns to add the two numbers into *R*5.

a. What is the minimum clock cycle time that can be used?

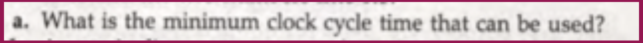
b. A nonpipeline system can perform the same operation by removing *R*3 and *R*4. How long will it take to multiply and add the operands without using the pipeline?

c. Calculate the speedup of the pipeline for 10 tasks and again for 100 tasks.

d. What is the maximum speedup that can be achieved?

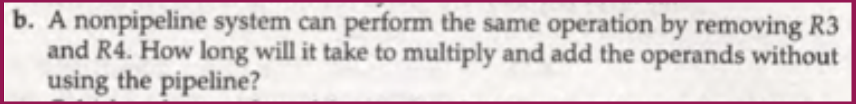


Solution:



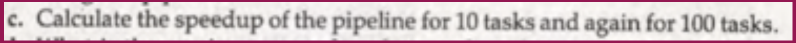
a. What is the minimum clock cycle time that can be used?

$t_p = 45 + 5 = 50$ ns $\quad (k = 3)$



b. A nonpipeline system can perform the same operation by removing *R3* and *R4*. How long will it take to multiply and add the operands without using the pipeline?

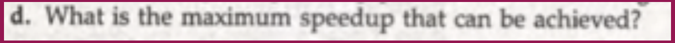
$t_n = 40 + 45 + 15 = 100$ ns



c. Calculate the speedup of the pipeline for 10 tasks and again for 100 tasks.

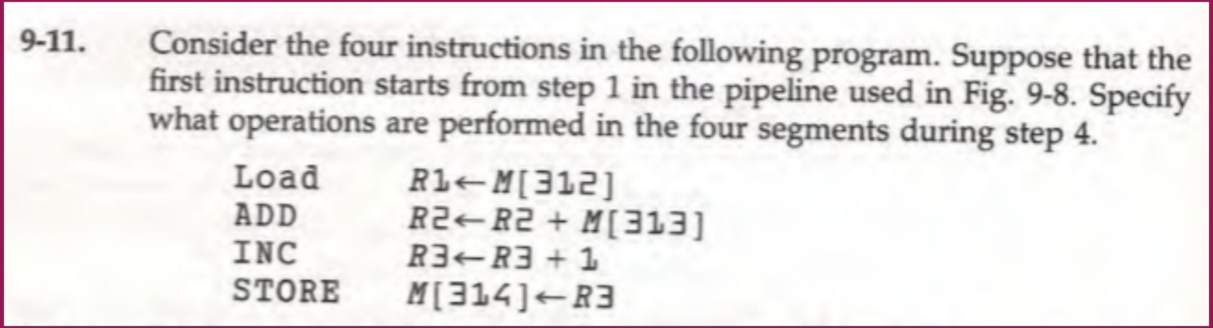
$S = n\, t_n / (k + n - 1)\, t_p = (10 \times 100) / [(3 + 9) \times 50] = 1.67$ (for $n = 10$)

$S = (100 \times 100) / [(3 + 99) \times 50] = 1.96$ (for $n = 100$)



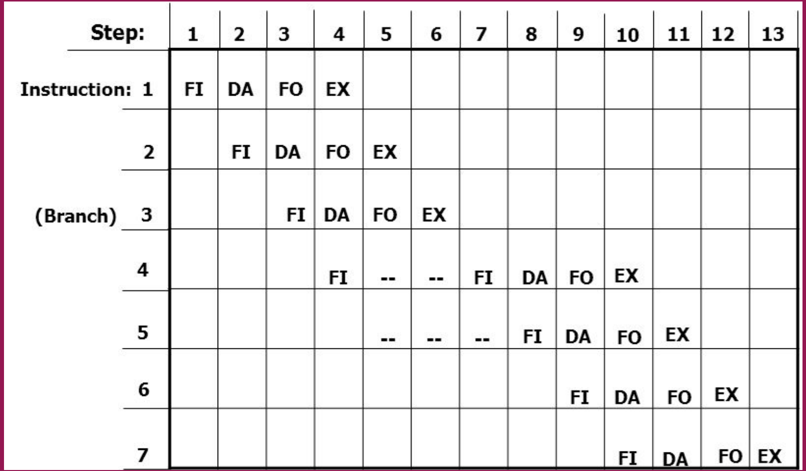
d. What is the maximum speedup that can be achieved?

$S_{max} = t_n / t_p = 100 / 50 = 2$



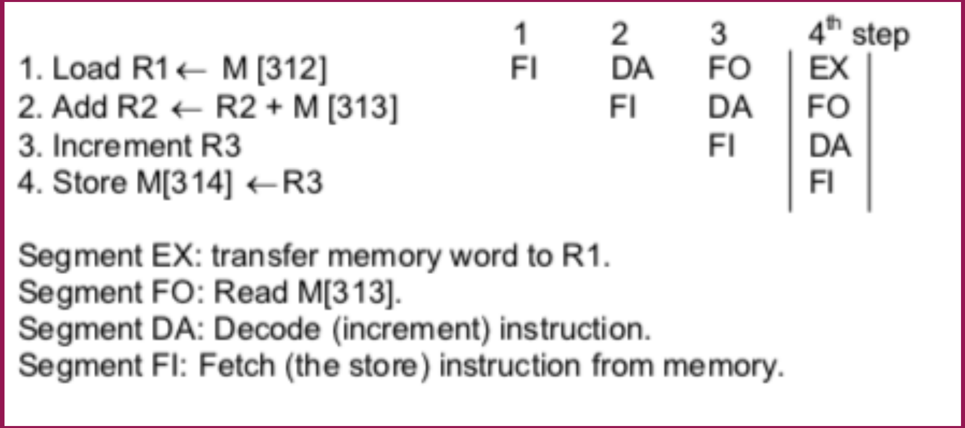
9-11. Consider the four instructions in the following program. Suppose that the first instruction starts from step 1 in the pipeline used in Fig. 9-8. Specify what operations are performed in the four segments during step 4.

```
Load    R1←M[312]
ADD     R2←R2 + M[313]
INC     R3←R3 + 1
STORE   M[314]←R3
```



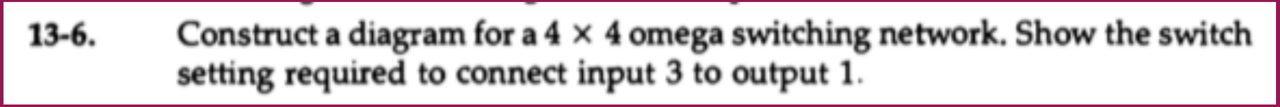
Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
(Branch) 3			FI	DA	FO	EX							
4				FI	--	--	FI	DA	FO	EX			
5					--	--	--	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Solution:

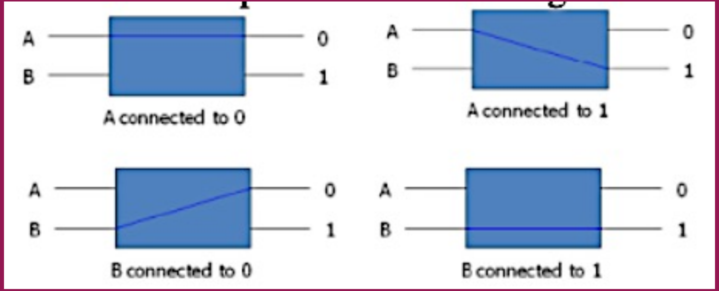


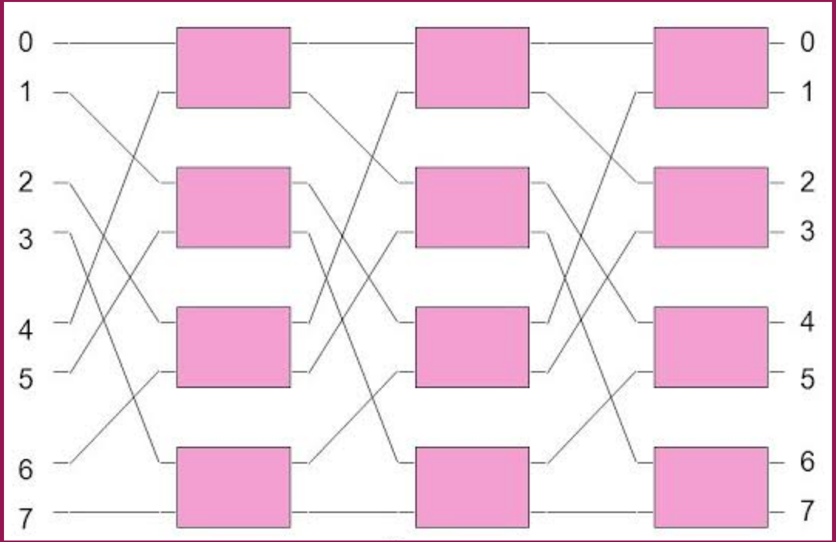
	1	2	3	4th step
1. Load R1 ← M [312]	FI	DA	FO	EX
2. Add R2 ← R2 + M [313]		FI	DA	FO
3. Increment R3			FI	DA
4. Store M[314] ← R3				FI

Segment EX: transfer memory word to R1.
Segment FO: Read M[313].
Segment DA: Decode (increment) instruction.
Segment FI: Fetch (the store) instruction from memory.



13-6. Construct a diagram for a 4 × 4 omega switching network. Show the switch setting required to connect input 3 to output 1.

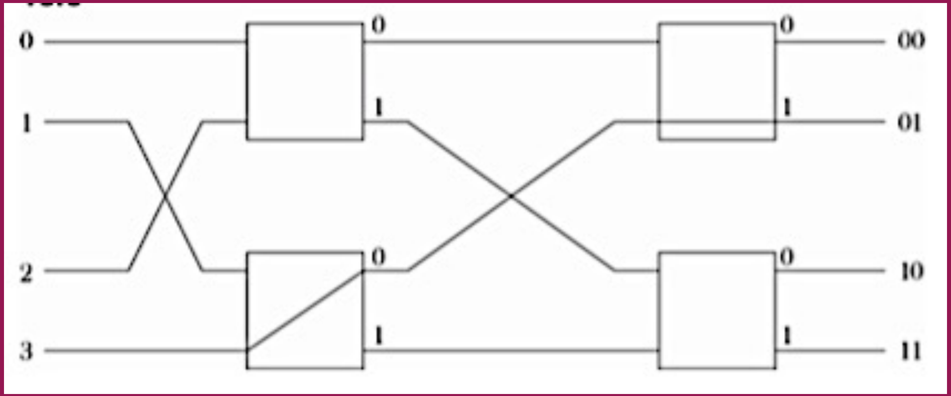


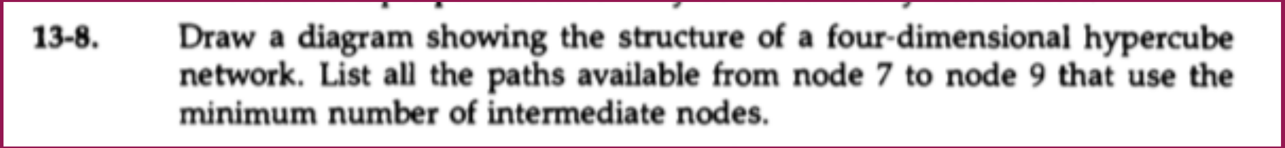


8 x 8 omega switching network

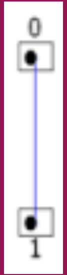
Solution:

Solution:

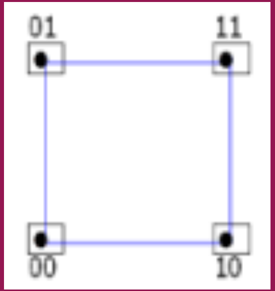




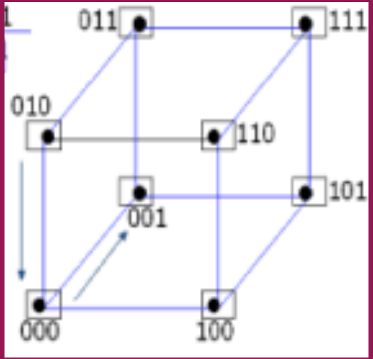
13-8. Draw a diagram showing the structure of a four-dimensional hypercube network. List all the paths available from node 7 to node 9 that use the minimum number of intermediate nodes.



One-Cube

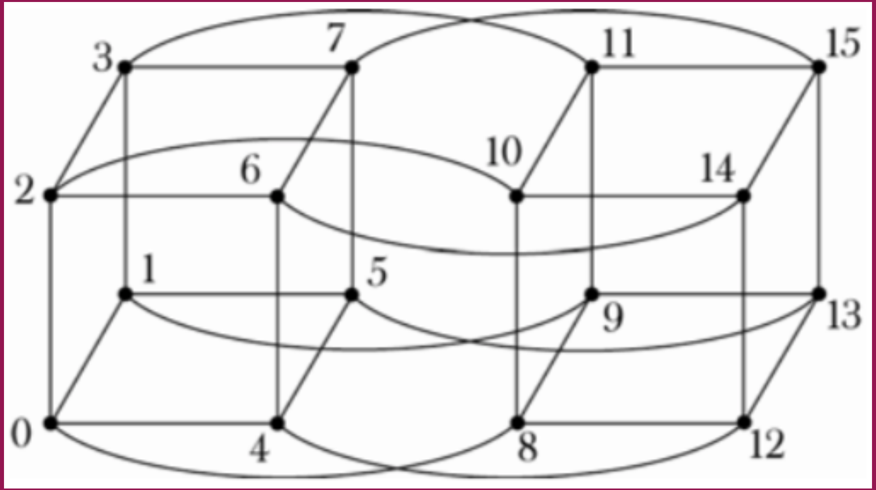


Two-Cube

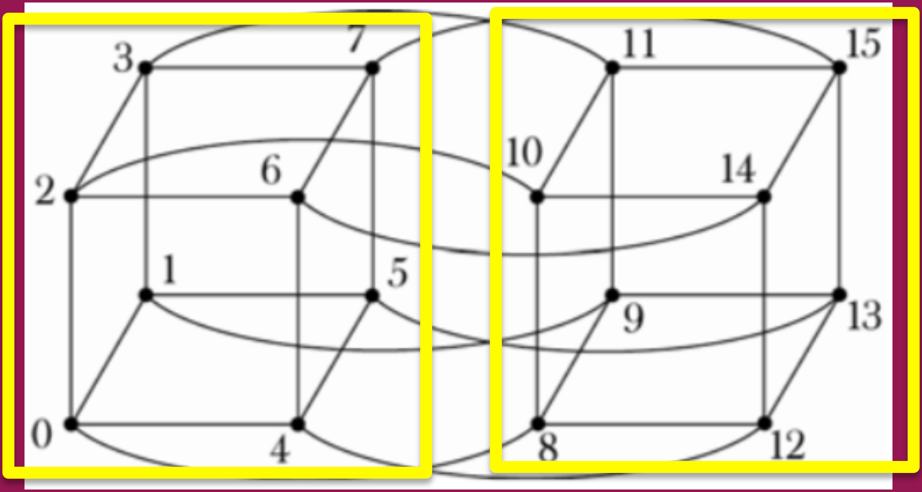


Three-Cube

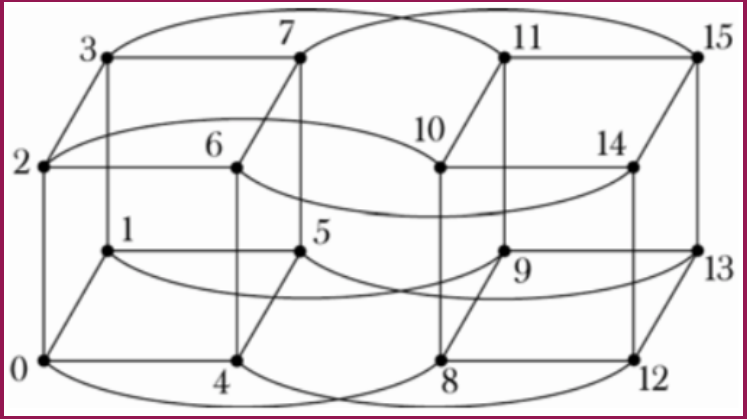
Solution:



Solution:



Solution:



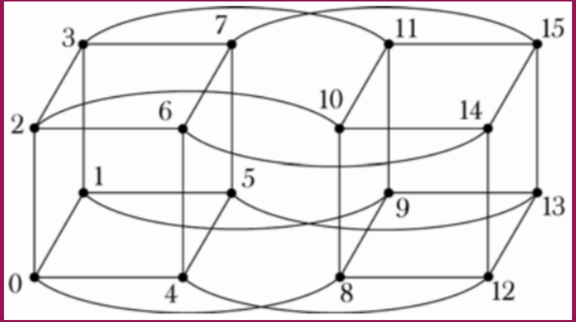
Paths available from node 7 to node 9 i.e., 0111 to 1001:

0111 XOR 1001 = 1110
So, we will have to traverse three axes (via two nodes).

Possible paths:

7 – 15 – 13 – 9
7 – 15 – 11 – 9
7 – 3 – 11 – 9
7 – 3 – 1 – 9
7 – 5 – 13 – 9
7 – 5 – 1 – 9

Solution:



Path 1: 7 – 15 – 13 – 9
(0111 – 1111 – 1101 – 1001)

Path 2: 7 – 15 – 11 – 9
(0111 – 1111 – 1011 – 1001)

Path 3: 7 – 3 – 11 – 9
(0111 – 0011 – 1011 – 1001)

Path 4: 7 – 3 – 1 – 9
(0111 – 0011 – 0001 – 1001)

Path 5: 7 – 5 – 13 – 9
(0111 – 0101 – 1011 – 1001)

Path 6: 7 – 5 – 1 – 9
(0111 – 0101 – 0001 – 1001)