



ECE-7th sem. CAO-Unit 6

Pipeline and Vector Processing

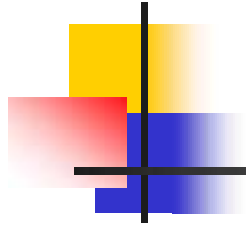
Dr.E V Prasad

12.10.17



Contents

- Parallel Processing
- Pipelining
- Arithmetic Pipeline
- Instruction Pipeline
- RISC Pipeline
- Vector Processing
- Array Processors



Parallel Processing



Introduction

- **Parallel processing** is a term used to denote a large class of techniques that are used to provide **simultaneous data-processing tasks** for the purpose of **increasing the computational speed** of a computer system.
- The purpose of parallel processing is to **speed up the computer processing capability** and **increase its throughput**, that is, the **amount of processing that can be accomplished during a given interval of time**.
- The amount of hardware increases with parallel processing, and with it, the cost of the system increases.



Introduction(cont.)

- Parallel processing can be viewed from various levels of complexity.
 - At the lowest level, we distinguish between parallel and serial operations by the type of registers used.
e.g. shift registers and registers with parallel load
 - At a higher level, it can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously.
- Shows one possible way of separating the execution unit into eight functional units (FUs or PEs or PU) operating in parallel.
 - A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

Processor with multiple functional units

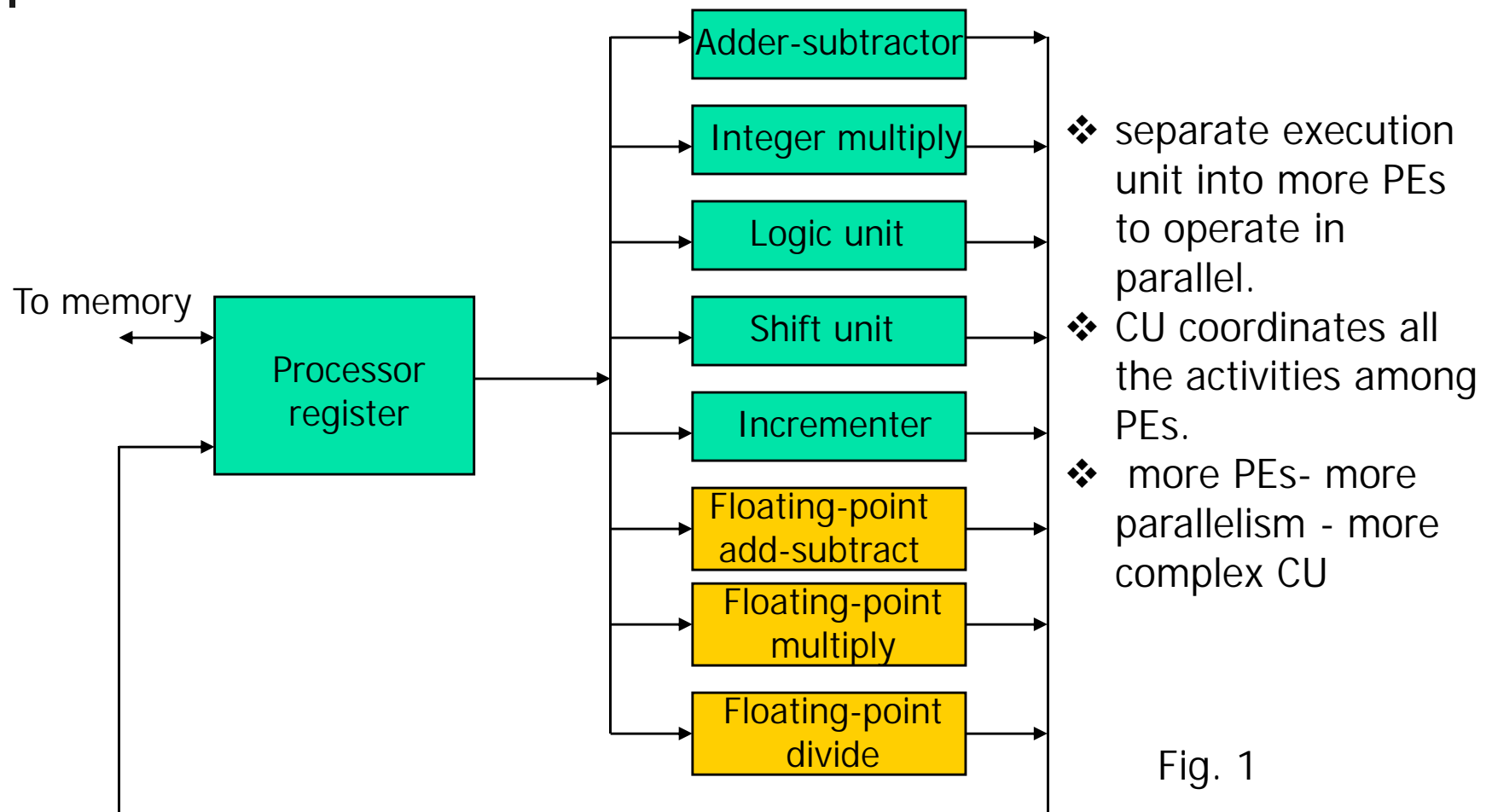


Fig. 1



Introduction(cont.)

- There are a variety of ways that parallel processing can be classified.
 - Internal organization of the processors
 - Interconnection structure between processors
 - The flow of information through the system
- M. J. Flynn considers the organization of a computer system by the number of instructions (instruction stream) and data items (data stream) that are manipulated simultaneously.



Architectural Classification

Flynn's classification

- Based on the multiplicity of *Instruction Streams* and *Data Streams*

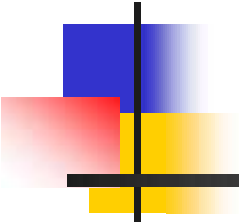
Instruction Stream

Sequence of Instructions read from memory

Data Stream

Operations performed on the data in the processor

PARALLEL COMPUTERS



		<i>Number of Data Streams</i>	
		<i>Single</i>	<i>Multiple</i>
<i>Number of Instruction Streams</i>	<i>Single</i>	SISD	SIMD
	<i>Multiple</i>	MISD	MIMD



SISD

- Represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- Instructions are executed **sequentially** and the system may or may not have internal parallel processing capabilities.
- parallel processing may be achieved by means of **multiple functional units** or by **pipeline processing**.



SIMD

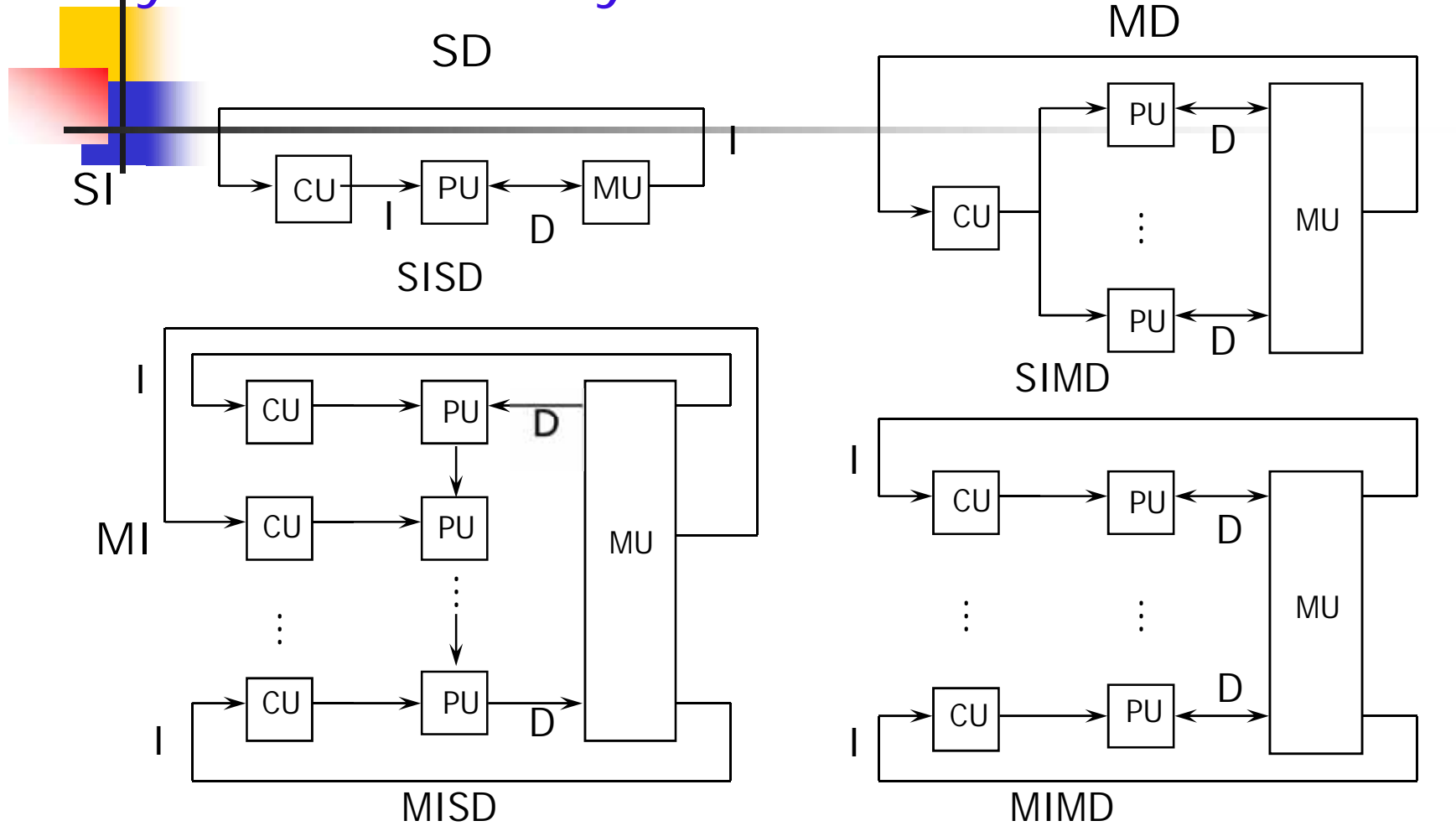
- Represents an organization that includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different items of data.
- The shared memory unit must contain **multiple modules** so that it can communicate with all the processors simultaneously.



MISD & MIMD

- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.
- MIMD organization refers to a computer system capable of processing several programs at the same time. e.g. multiprocessor and multicomputer system
- Flynn's classification depends on the distinction between the performance of the control unit and the data-processing unit.
 - It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections.

Flynn's taxonomy

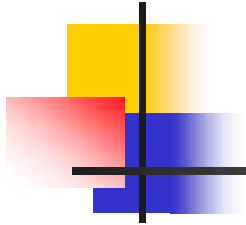


CU- control unit ; PU - processing unit (ALU) ; MU - memory Unit



Introduction(cont.)

- One type of parallel processing that does not fit Flynn's classification is pipelining.
- We consider parallel processing under the following main topics:
 - Pipeline processing
 - Is an implementation technique where arithmetic sub-operations or the phases of a computer instruction cycle overlap in execution.
 - Vector processing
 - Deals with computations involving large vectors and matrices.
 - Array processing
 - Perform computations on large arrays of data.



Pipelining



Pipelining

- Pipelining is a technique of decomposing a sequential process into sub-operations (segments)
- Divide the processor into segment processors each one is dedicated to a particular segment.
- Each segment is executed in a dedicated segment-processor operates concurrently with all other segments.
- Information flows through these multiple hardware segments.
- The overlapping of computation is made possible by associating a register with each segment in the pipeline.
- The registers provide isolation between each segment so that each can operate on distinct data simultaneously

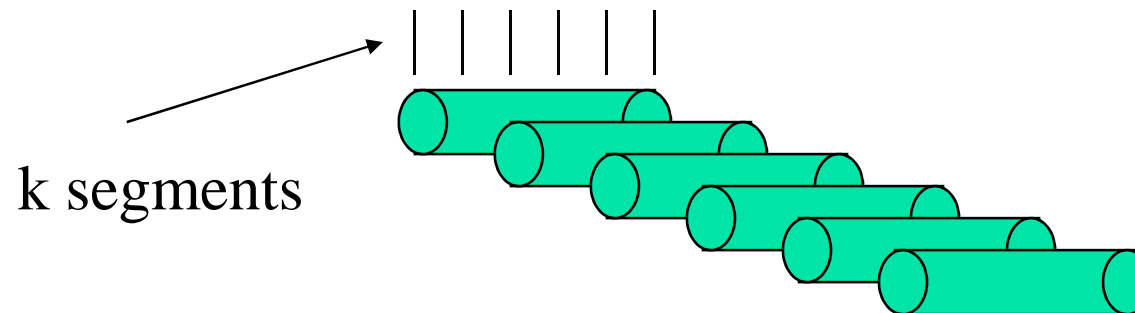


Pipelining(cont.)

- Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an **input register** followed by a **combinational circuit**.
 - The register holds the data.
 - The combinational circuit performs the sub-operation in the particular segment.
- A clock is applied to all registers after **enough time** has elapsed to perform all segment activity.

Pipelining

- Instruction execution is divided into k segments or stages
 - Instruction exits pipe stage $k-1$ and proceeds into pipe stage k
 - All pipe stages take the same amount of time; called one processor cycle
 - Length of the processor cycle is determined by the **slowest** pipe stage

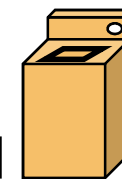
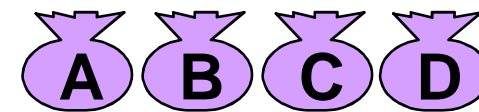


Pipelining: Laundry Example

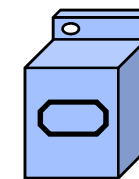
- Small laundry has one washer, one dryer and one operator.

- ❖ Washer takes 30 minutes
- ❖ Dryer takes 40 minutes
- ❖ "operator folding" takes 20 minutes
- ❖ Assume 4 loads /tasks
- ❖ It takes (t_n) 90 minutes to complete one load
- ❖ In other words he will not start a new task unless he is already done with the previous task.
- ❖ The process is sequential.
- ❖ Sequential laundry takes 6 hours for 4 loads.

4 loads/tasks



Washer

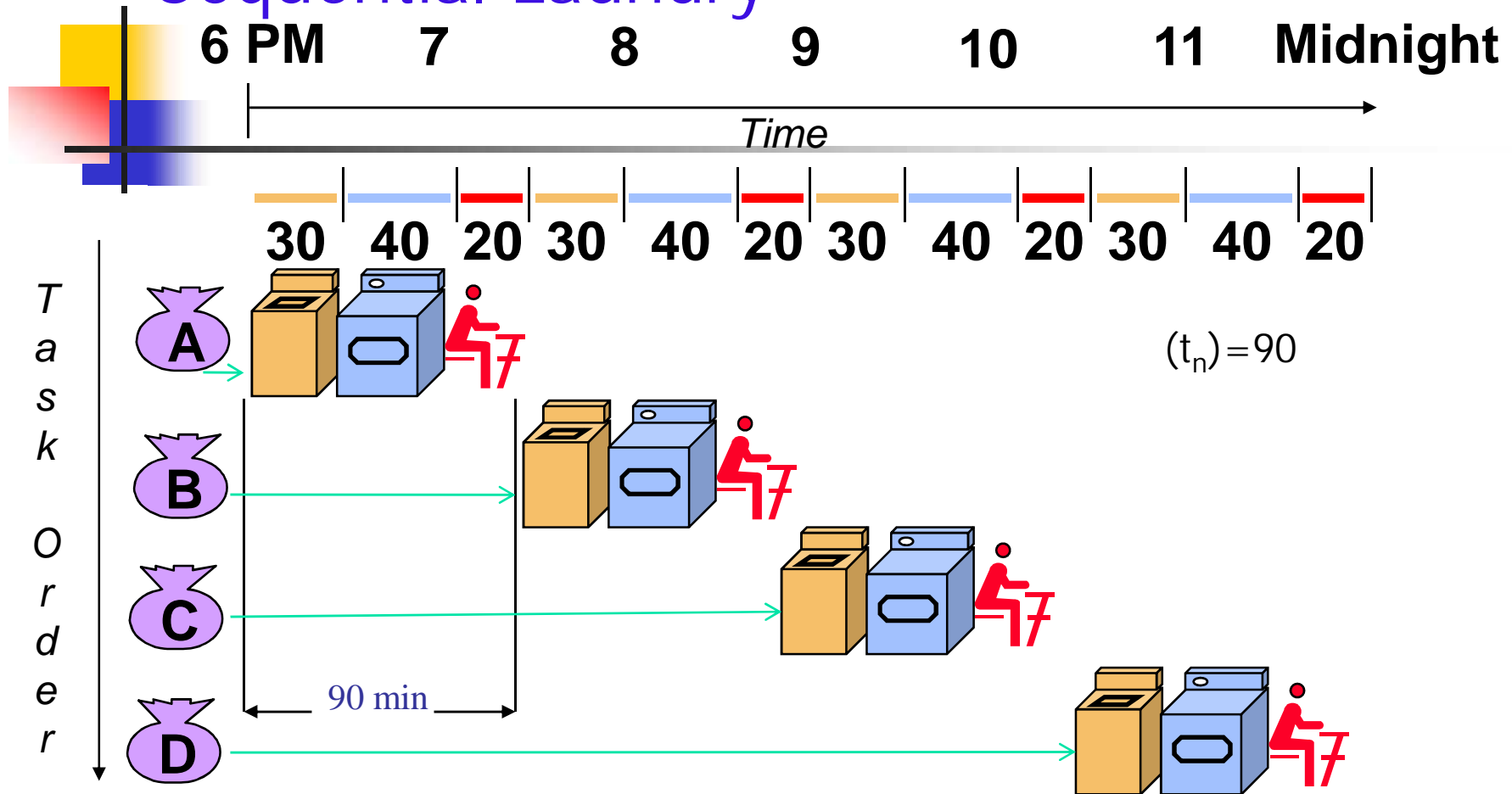


Dryer



Operator folding

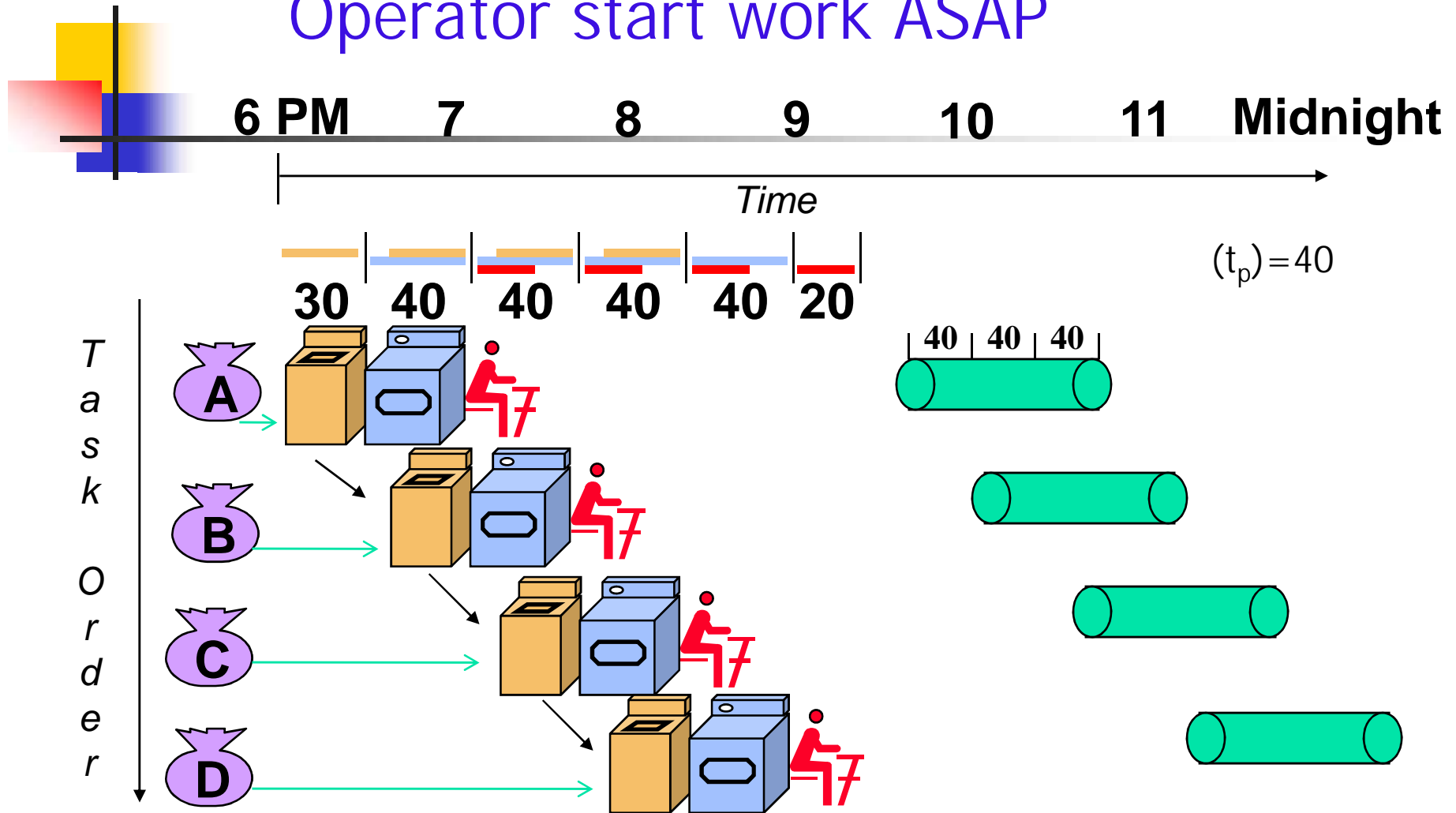
Sequential Laundry



- A better idea would be start the next load washing while the first is drying.
- Then, while the first load was being folded, the second load would dry and a new load could be put in the washer'

Efficiently scheduled laundry: Pipelined Laundry

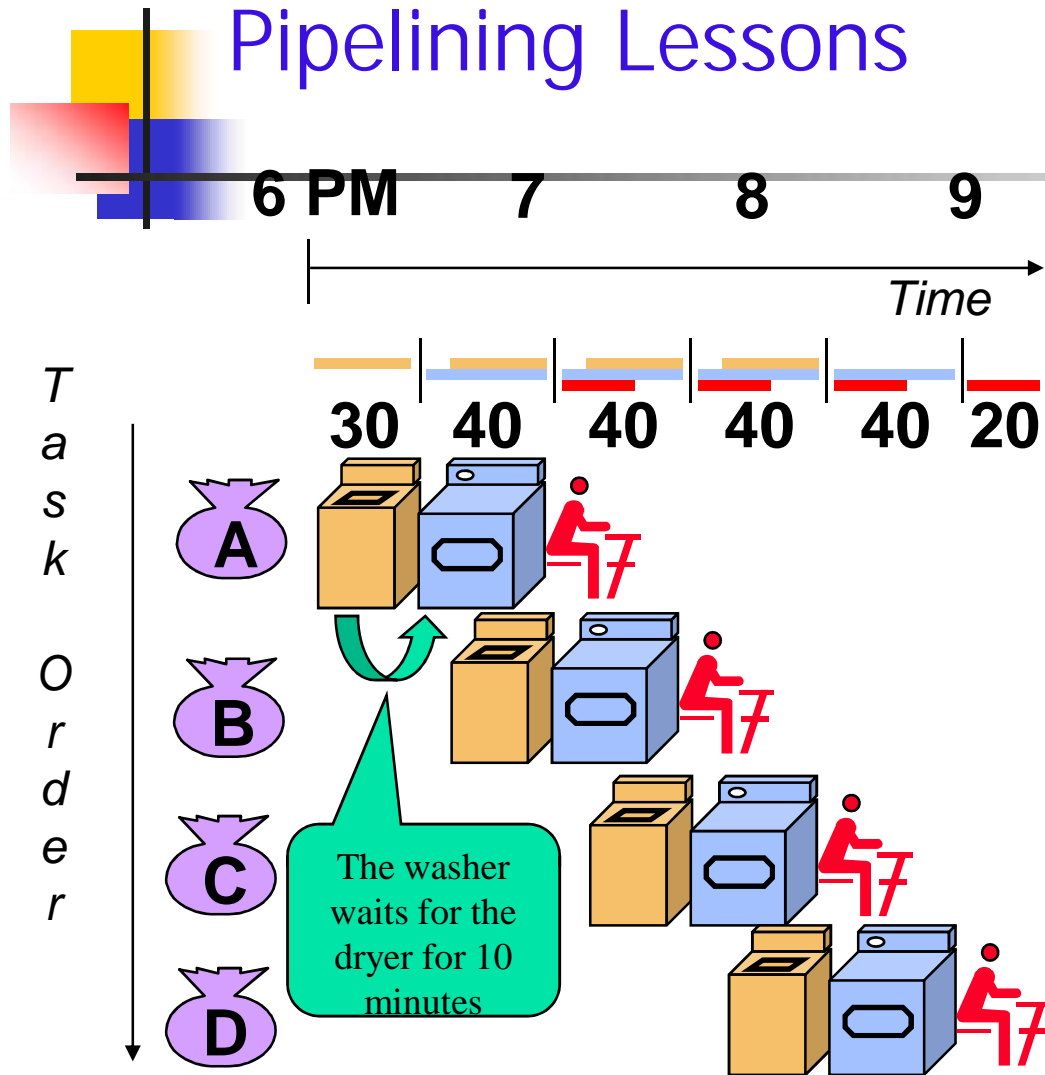
Operator start work ASAP



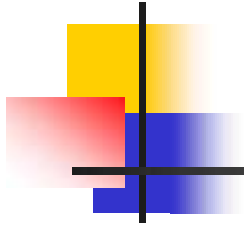
Pipeline Laundry

- Operator asks for the delivery of loads to the laundry every $(t_p)40$ minutes!?
- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



- Multiple tasks operating simultaneously.
- Pipelining doesn't help latency of single task, it helps throughput of entire workload.
- Pipeline rate limited by slowest pipeline stage.
- Potential speedup = Number of pipe stages.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to "fill" pipeline and time to "drain" it reduces speedup.
- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of any individual instruction.



- The pipeline organization will be demonstrated by means of a simple example.
 - To perform the combined multiply and add operations with a stream of numbers
$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \dots, 7$$



Pipeline organization

- Each suboperation is to be implemented in a segment within a pipeline.

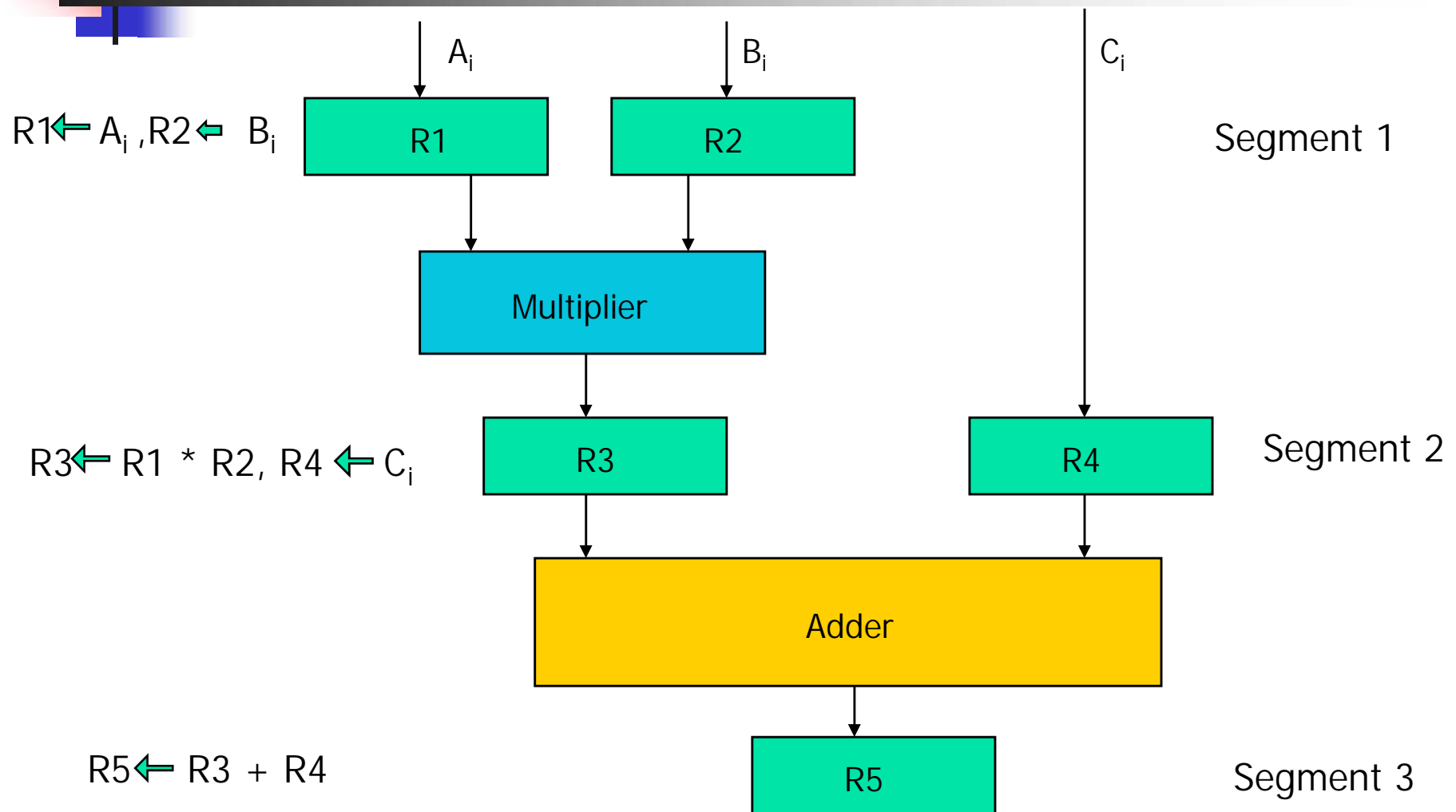
$R1 \leftarrow A_i, R2 \leftarrow B_i$ Input A_i and B_i

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

$R5 \leftarrow R3 + R4$ Add C_i to product

- Each segment has one or two registers and a combinational circuit as shown in Fig.2.
- The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 1.

Example of pipeline processing





Content of registers in pipeline example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	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	--	--	--	--	$A_7 * B_7 + C_7$



General considerations

- Any operation that can be decomposed into a sequence of sub-operations of about the **same complexity** can be implemented by a pipeline processor.
- The general structure of a four-segment pipeline is illustrated in Fig.3.
- We define a **task** as the total operation performed going through all the segments in the pipeline.
- The behavior of a pipeline can be illustrated with a **space-time** diagram.
 - It shows the segment utilization as a function of time.



Design of a basic pipeline

- ❖ Pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that output of one stage is connected to input of next stage and each stage performs a specific operation.
- ❖ Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.
- ❖ All the stages in the pipeline along with the interface registers are controlled by a common clock.

4-segment pipeline

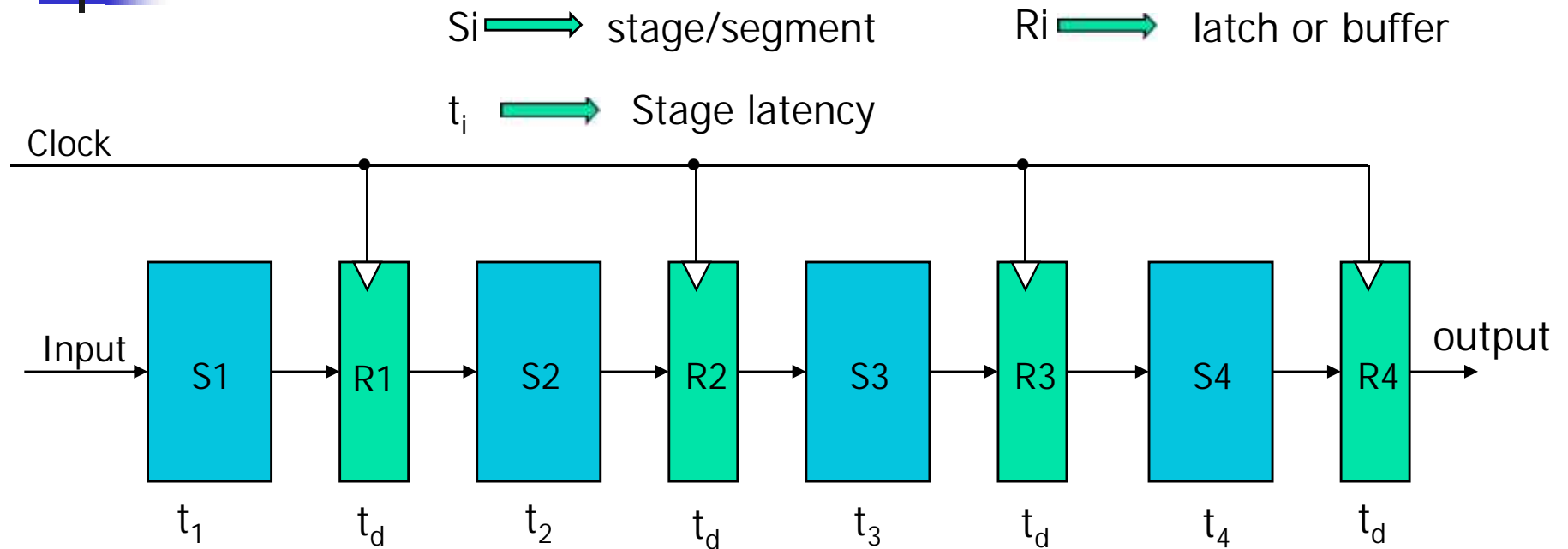
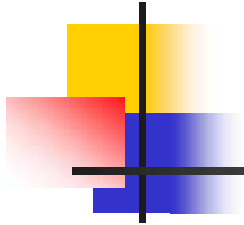


Fig.3

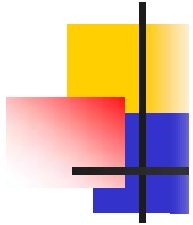
what is the clock cycle time (t_p)?

Latch delay : t_d ; $t_p = \max \{t_i\} + t_d$; Pipeline frequency : $f = 1 / t_p$



-
- The space-time diagrams for execution of 2 instructions with :
 - i) a four-segment pipeline
 - ii) a four-segment non-pipelineare demonstrated in Fig.4 (a).





Non overlapped execution: *Non-pipeline*

<u>Stage\Cycle</u>	1	2	3	4	5	6	7	8
S1	I ₁				I ₂			
S2		I ₁				I ₂		
S3			I ₁				I ₂	
S4				I ₁				I ₂

Total time = 8 cycles

Overlapped execution: *pipeline*

<u>Stage\Cycle</u>	1	2	3	4	5
S1	I ₁	I ₂			
S2		I ₁	I ₂		
S3			I ₁	I ₂	
S4				I ₁	I ₂

Total time = 5 cycles

Fig.4 space-time diagrams



Problem

Consider a k (4) -segment pipeline , with a clock cycle time t_p , used to execute n (6) tasks.

-Draw the space-time diagram and observe:

- i) The utilization of the segments.
- ii) Time of execution of single task, two tasks and so
- iii) Efficiency of the pipeline



Space-time diagram for pipeline

task $\Rightarrow T_i$

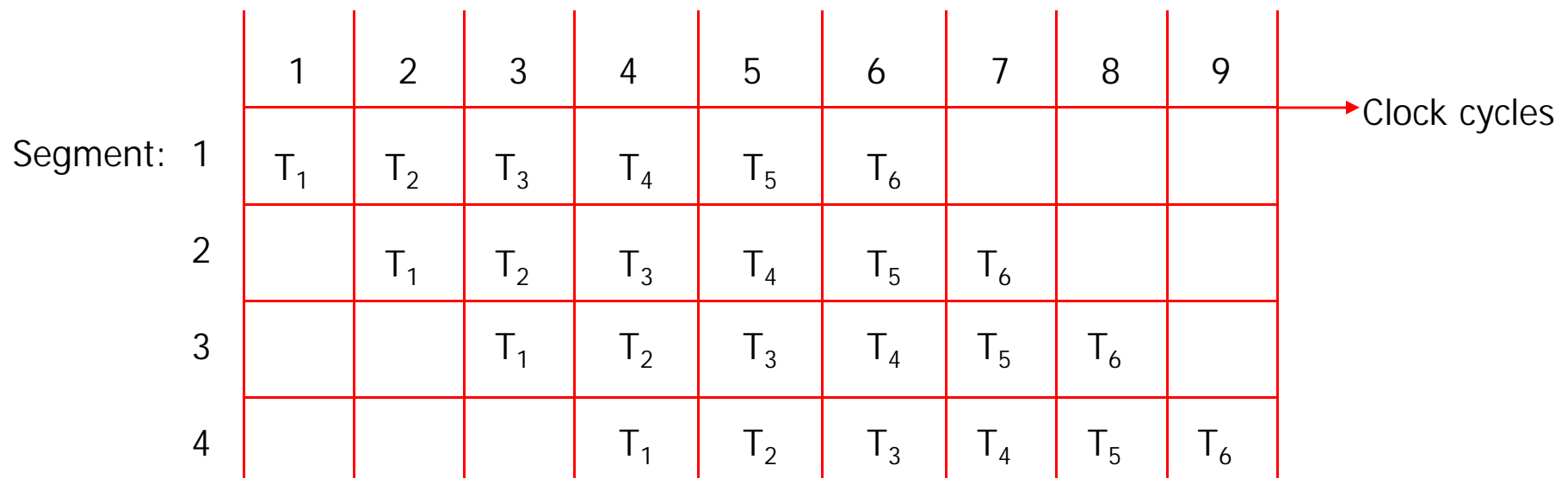


Fig. space-time diagram

Speedup ?
Efficiency ?



Pipeline Speedup

- Consider a k-segment pipeline ,with a clock cycle time t_p , to execute n tasks.
 - ❖ The first task T_1 requires a time equal to kt_p to complete its operation.
 - ❖ The remaining (n-1) tasks will be completed after a time equal to $(n-1)t_p$
 - ❖ Therefore, to complete n tasks using a k-segment pipeline requires $k+(n-1)$ clock cycles.
- Consider a non-pipeline unit that performs the same operation and takes a time equal to t_n to complete each task.
 - The total time required for n tasks is nt_n .



Introduction(cont.)

- The **speedup** of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

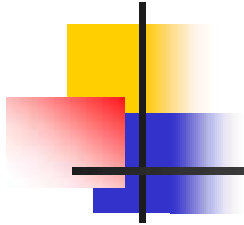
$$S_k = nt_n / (k+n -1)t_p$$

- If n becomes much larger than $(k-1)$, the speedup becomes $S_k = t_n / t_p$.
- If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, i.e., $t_n = kt_p$, the speedup reduces to $S_k = k t_p / t_p = k$.
- This shows that the theoretical maximum speedup that a pipeline can provide is k , where k is the number of segments in the pipeline.



Problem

- The following numerical example may clarify the sub-operations performed in each segment (Fig.6)
- The comparator, shift, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits.
- Suppose that the time delays of the four segments are $t_1=60\text{ns}$, $t_2=70\text{ns}$, $t_3=100\text{ns}$, $t_4=80\text{ns}$, and the interface registers have a delay of $t_r=10\text{ns}$
 - Pipeline floating-point arithmetic delay: $t_p=t_3+t_r=110\text{ns}$
 - Non-pipeline floating-point arithmetic delay:
 $t_n=t_1+t_2+t_3+t_4+t_r=320\text{ns}$
 - Speedup: $320/110=2.9$



- To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel.
- This is illustrated in Fig.5, where four identical circuits are connected in parallel.
- Instead of operating with the **input data in sequence** as in a pipeline, the parallel circuits accept four input data items **simultaneously** and perform four tasks at the same time.



Speed-up

For e.g., if a pipeline has 4 stages and 5 inputs

Speedup over an equivalent non-pipeline processing ?

What is the maximum value of speedup ?

$$S_k = nk / (k+n -1)$$

The speedup value increases with increased number of tasks .

When the number of tasks 'n' are significantly larger than k,
that is, $n \gg k$

Lt $[\text{Speedup}] = K$, reaches max value of S_k
 $n \rightarrow$



Efficiency and Throughput

Efficiency of the k-stages pipeline :

Indicator of how efficiently the resources of the pipeline are used.

$$\text{Efficiency} = E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)}$$

Efficiency = Given speed up / Max speed up = $S_k / S_k(\text{max})$

What is the maximum value of efficiency ? When ...?

What is the lowest value of efficiency? When ...?

Pipeline throughput (also called bandwidth)

It is the measure of how often an instruction exits the pipeline.

It is the average number of results computed in one cycle.

Or the number of input tasks that can be performed in one cycle time.

$$H_k = \frac{n}{[k + (n-1)] \dagger} = \frac{n f}{k + (n-1)}$$

i.e Number of instructions / Total time to complete the instructions

What is the maximum value of H_k ?



Note

- E_k , the speedup per stage, approaches its maximum value of 1 when $n \rightarrow \infty$.
- When $n=1$, E_k will have the value $1/k$, which is the lowest obtainable value
- When $n \rightarrow \infty$, the throughput H_k approaches the maximum value of one task per clock cycle.
- Pipeline throughput = Efficiency * frequency



Latency, Efficiency and Throughput

Pipeline latency

- ❖ Each instruction takes a certain time to complete.
- ❖ This is the latency for that operation.
- ❖ It's the amount of time between when the instruction is issued and when it completes.

Pipeline Throughput

- ❖ the rate at which operations get executed-
- ❖ generally expressed as operations/second or operations/cycle
- ❖ Need not be the same as dividing the time span by the latency

In machines with no pipelining:

- ❖ The machine cycle must be long enough to complete a single instruction
Throughput = $1/\text{latency}$, The latency is the same as cycle time
Since each operation executes by itself
- ❖ If an instruction is divided into smaller chunks (multiple clock cycles per instruction) then Throughput is not the same



Problem 1

Consider the execution of a program of 10/100/15000 instructions by a linear pipeline processor with a clock rate of 25MHz. Assume that the instruction pipeline has 5 stages and that one instruction is issued per clock cycle. The penalties due to branch instructions and out-of-sequence executions are ignored

- a) Calculate the speedup factor as compared with non-pipelined processor
- b) What are the efficiency and throughput of this pipelined processor?



From the problem:

$n = 10,100$ and 15000

$f = 25\text{MHz}$

$K = 5$ stages

$$\text{III) Speedup } S_k = (nk) / (n+k-1) = (1500 \cdot 5) / (1500+5-1) \\ = 75,000 / 1504 = 4.999$$

$$\text{Efficiency} = E_k = S_k / k = 4.999 / 5 = 0.999$$

$$\text{Throughput} = H_k = 0.999 \cdot 25 \text{ MHz} = 24.99 \text{ MIPS}$$

$$\text{II) } S_k = 500 / 104 = 4.8076 ; E_k = 4.8076 / 5 = 0.9615 ; H_k = 24.038 \text{ MIPS}$$

$$\text{I) } S_k = 50 / 14 = 3.5714 ; E_k = 3.5714 / 5 = 0.7143 ; H_k = 17.857 \text{ MIPS}$$

Problem2.

Consider 5 stages of the processors that have the following latencies (in p sec.) Assume that when pipelining, each pipeline stage costs 20ps extra for the registers between pipeline stages.

	Fetch	Decode	Cal. operand address	Execute	Save results
Pipeline 1	300	400	350	550	100
pipeline 2	200	150	100	190	140

- 1) For both Non-pipelined and pipelined processing, Compute :
what is the cycle time? What is the latency of an instruction?
What is the throughput?
- 2) If you could split one of the pipeline stages into 2 equal halves, which one would you choose? What is the new cycle time? What is the new latency? What is the new throughput?



1 (a) Non-pipelined processing;

Because there is no pipelining, the cycle time must allow an instruction to go through all stages in one cycle.

The latency is the same as cycle time since it takes the instruction one cycle to go from the beginning of fetch to the end of writeback (save).

The throughput is defined as $1/CT$ inst/s.

$$P 1 : CT = 300 + 400 + 350 + 550 + 100 = 1700ps$$

$$\text{Latency} = 1700ps$$

$$\text{Throughput} = 1/1700 \text{ inst/ps}$$

$$P 2 : CT = 200 + 150 + 100 + 190 + 140 = 780ps$$

$$\text{Latency} = 780ps$$

$$\text{Throughput} = 1/780 \text{ inst/ps}$$



1 (b) Pipelined processing:

Pipelining reduces the cycle time to the length of the longest stage plus the register delay.

Latency becomes $CT \cdot N$ where N is the number of stages as one instruction will need to go through each of the stages and each stage takes one cycle.

The throughput formula remains the same.

$$P 1. CT = 550 + 20 = 570 \text{ ps}$$

$$\text{Latency} = 5 * 570 = 2850\text{ps}$$

$$\text{Throughput} = 1/570 \text{ inst/ps}$$

$$P 2. CT = 200 + 20 = 220 \text{ ps}$$

$$\text{Latency} = 5 * 220 = 1100\text{ps}$$

$$\text{Throughput} = 1/220 \text{ inst/ps}$$



2. If you could split one of the pipeline stages into 2 equal halves.

We would want to choose the longest stage to split into half. The new cycle time becomes the originally 2nd longest stage length. Calculate latency and throughput correspondingly, but remember there are now 6 stages instead of 5.

a. $CT = 400 + 20 = 420 \text{ ps}$

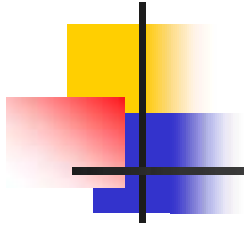
$$\text{Latency} = 6 * 420 = 2520 \text{ ps}$$

$$\text{Throughput} = 1/420 \text{ inst/ps}$$

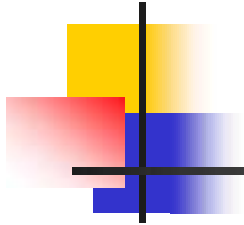
b. $CT = 190 + 20 = 210 \text{ ps}$

$$\text{Latency} = 6 * 210 = 1260 \text{ ps}$$

$$\text{Throughput} = 1/210 \text{ inst/ps}$$



How the performance of a
pipeline be improved ?



How to improve the performance of a pipeline?

1. Make the clock rate faster.
2. Duplicate functional units to allow parallel execution of instructions.
3. Increase the number of stages in the pipeline
4. Allow all pipeline stages possibly take the same process time.

Ideally, all stages should be exactly the same length.

5. By avoiding data dependency/unconditional jumps



Multiple functional units in parallel

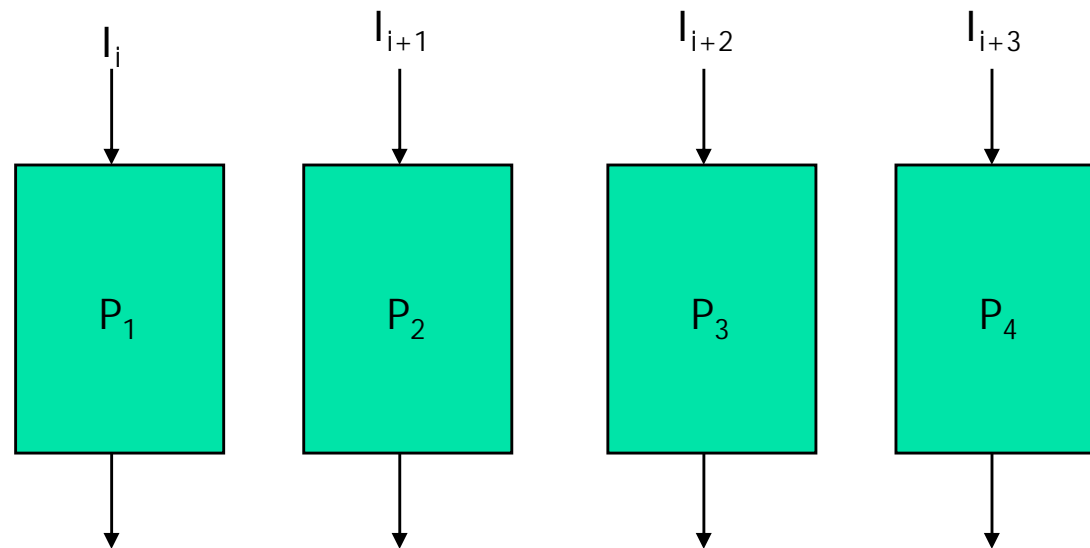
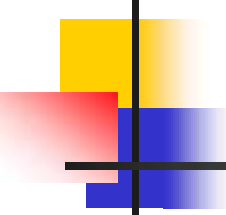
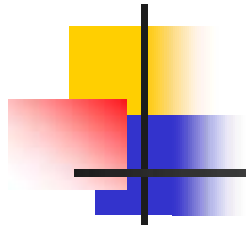


Fig.5

- 
-
- There are various reasons why the pipeline cannot operate at its maximum theoretical rate.
 - Different segments may take different times to complete their suboperation.
 - It is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit.
 - There are two areas of computer design where the pipeline organization is applicable.
 - Arithmetic pipeline
 - Instruction pipeline



Arithmetic Pipeline



Arithmetic pipeline - Introduction

- Applications of arithmetic Pipeline units are usually found in very high speed computers
 - Floating–point operations, multiplication of fixed-point numbers, and similar computations in scientific problem
- Floating–point operations are easily decomposed into sub-operations as demonstrated in Sec.5.
- Application of a pipeline unit for floating-point addition and subtraction is showed in the following:
 - The inputs to the floating-point adder pipeline are two normalized floating-point binary number

$$\begin{aligned} X &= A \times 2^a \\ Y &= B \times 2^b \end{aligned}$$

A and B are two fractions that represent the mantissas
a and b are two integers that represent the the exponents



Floating-point addition

The floating-point addition and subtraction can be performed in four segments, as shown in Fig.6.

- The suboperations that are performed in the four segments are:
 - **Compare the exponents**
 - The larger exponent is chosen as the exponent of the result.
 - **Align the mantissas**
 - The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right.
 - **Perform the operation (Add or subtract the mantissas)**
 - **Normalize the result**
 - When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one.
 - If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.



Floating Point Adder Unit

- Our purpose is to compute the sum

$$F = A + B = c \times 10^r = d \times 10^s$$

where $A = a \times 10^p$; $B = b \times 10^q$

$$r = \max(p, q) \text{ and } 0.1 \leq d < 1$$

- For example:

$$A = 0.9504 \times 10^3$$

$$B = 0.8200 \times 10^2$$

$$a = 0.9504 \quad b = 0.8200$$

$$p = 3 \quad \& \quad q = 2 ; r = 3$$



Floating Point Adder Unit

Operations performed in the four pipeline stages are :

1. Compare the exponents

Compare p and q and choose the largest exponent,

$r = \max(p, q)$ and

compute difference of exponents: $t = |p - q|$

Example:

$$r = \max(p, q) = 3$$

$$t = |p - q| = |3 - 2| = 1$$



Floating Point Adder Unit

2. Align the mantissas

Rewrite the smaller number such that its exponent matches with the exponent of the larger number.

Shift right the fraction associated with the smaller exponent by t units to equalize the two exponents before fraction addition.

- Example:

Smaller exponent, $b = 0.8200$

Shift right b by 1 ($=t$) unit is 0.082



Floating Point Adder Unit

3. Perform the operation

Perform fixed-point addition of two fractions to produce the intermediate sum fraction c

■ Example :

$$a = 0.9504 ; b = 0.082$$

$$c = a + b = 0.9504 + 0.082 = 1.0324$$



Floating Point Adder Unit

4. Normalize the result-(align radix point)

(shift mantissa and adjust exponent)

Count the number of leading zeros (u) in fraction c and shift left c by u units to produce the normalized fraction $d = c \times 10^u$, with a leading bit 1.

Update the large exponent s by subtracting $s = (r - u)$ to produce the output exponent.

■ Example:

$$c = 1.0324 = 0.10324 \times 10^{(3+1)}$$

$u = -1 \rightarrow$ right shift

$$d = 0.10324, s = r - u = 3 - (-1) = 4$$

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

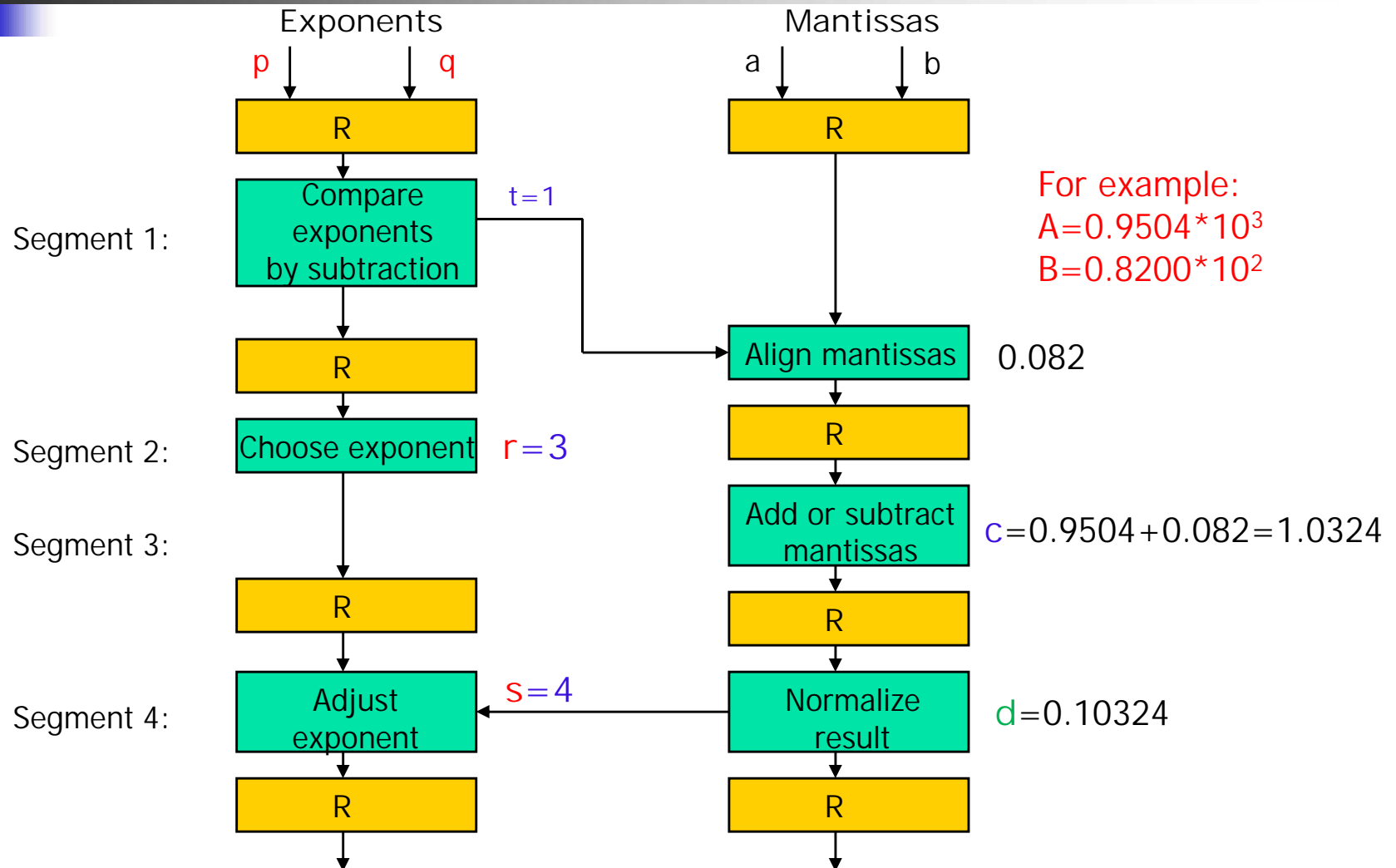


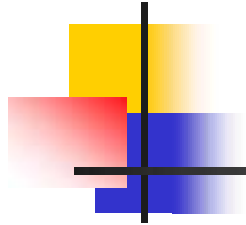
Floating Point Adder Unit

FP adder/subtractor is implemented by using combinational logic circuits in the following 4 stages:

1. Comparator / Subtractor
2. Shifter
3. Fixed Point Adder-cum-subtractor
4. Normalizer (leading zero counter and shifter)

Pipeline for floating-point addition/subtraction





Instruction Pipeline



Instruction Pipeline

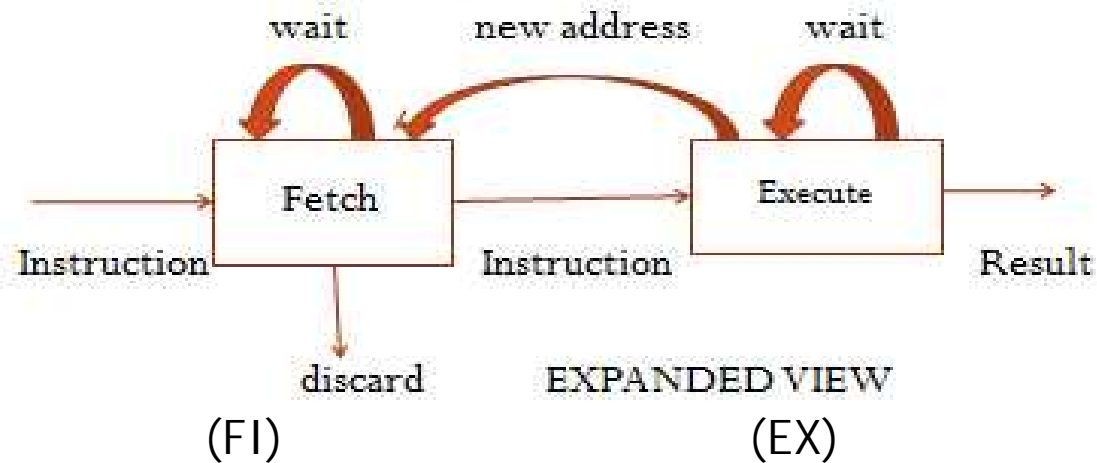
- Pipeline processing can occur not only in the **data stream** but in the **instruction** as well.
- Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a **two-segment** pipeline.
- Computers with complex instructions require other phases in addition to above phases to process an instruction completely.



Two stage instruction pipelining



Simplified view





6 segment Instruction pipeline

1. Fetch the instruction from memory (FI)

Read next instruction into CPU

2. Decode the instruction (DI)

Determine opcode and operand specifiers.

3. Calculate the effective address (CA)

Calculate the effective addresses of all operands
(on branch, calculate target address of branch)

4. Fetch the operands (FO)

Fetch operands from memory or register file

5. Execute the instruction (EX)

Perform the indicated operation

6. Store the result (SR)

Write operand to memory or register file

❖ 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

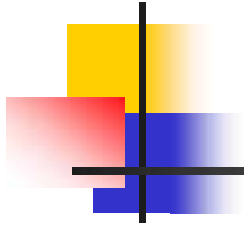


4 segment pipeline

- FI: segment 1 that fetches the instruction.
- DA: segment 2 that decodes the instruction and calculates the effective address.
- FO: segment 3 that fetches the operands.
- EX: segment 4 that executes the instruction.

Step	1	2	3	4	5	6	7	8	9
1	FI	DA	FO	EX					
2		FI	DA	FO	EX				
3			FI	DA	FO	EX			
4				FI	DA	FO	EX		
5					FI	DA	FO	EX	
6						FI	DA	FO	EX

Fig: timing diagram for 4-segment instruction pipeline



Draw the timing diagrams for:

- 2 - stage instruction pipeline
- 3 - stage instruction pipeline
- 6 - stage instruction pipeline



Limitations at glance

- There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate.
 - Different segments may take different times to operate on the incoming information.
 - Some segments are skipped for certain operations.
 - Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory



Example: four-segment instruction pipeline

- Assume that:
 - The decoding of the instruction can be combined with the calculation of the effective address into one segment.
 - The instruction execution and storing of the result can be combined into one segment.
- Fig.7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline.
 - Thus up to four sub-operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Four-segment pipeline

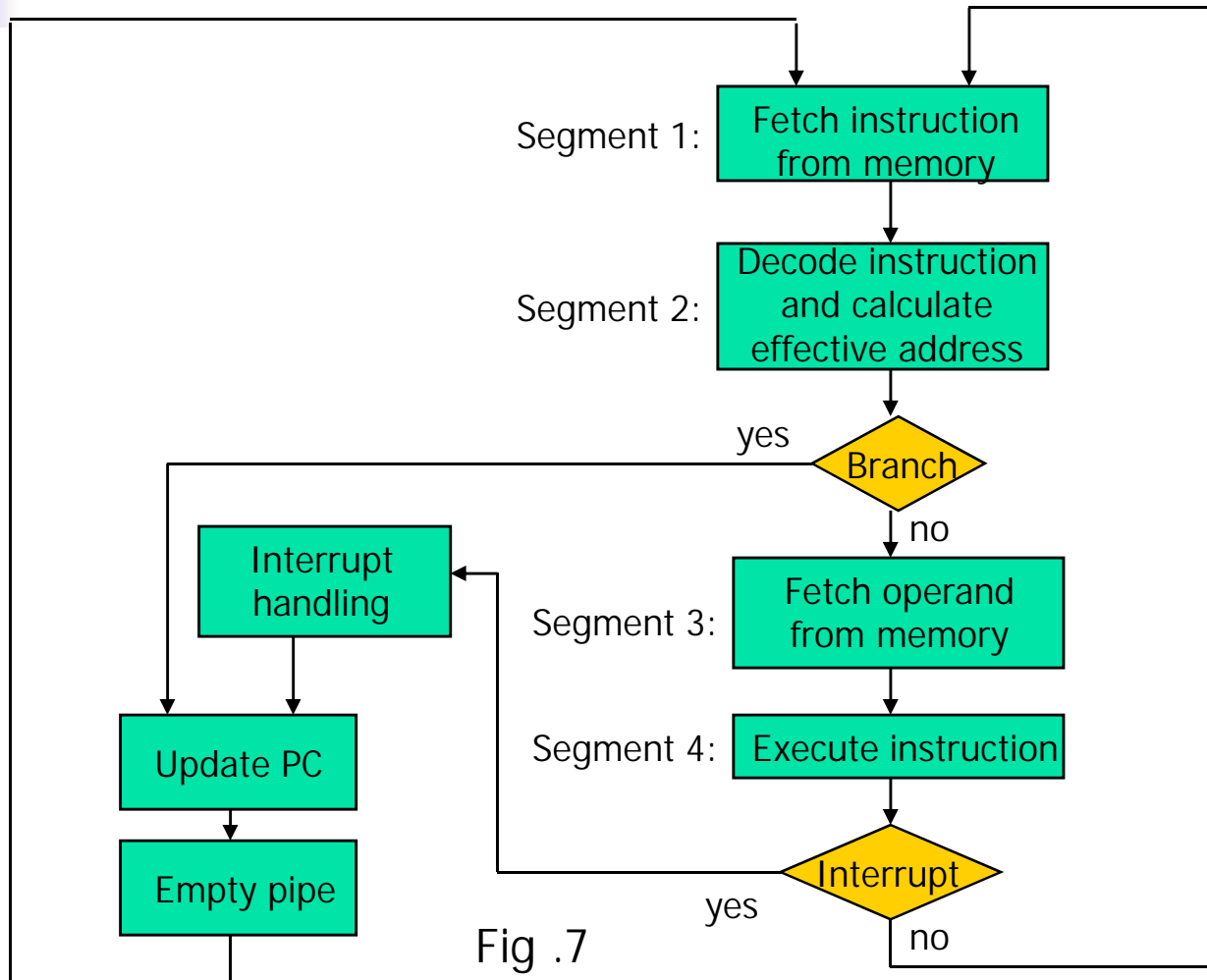
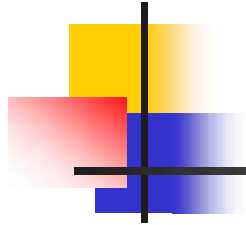


Fig .7



- An instruction in the sequence 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 instruction buffer is deleted.
 - Similarly, an interrupt request will cause the pipeline to empty and start again from a new address value.
- Fig.8 shows the operation (Timing of instruction pipeline) of the instruction pipeline.

Timing of instruction pipeline

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

FI: the segment that fetches an instruction

DA: the segment that decodes the instruction and calculate the effective address

FO: the segment that fetches the operand

EX: the segment that executes the instruction

Fig.8



Pipelining Hazards

Hazard: A condition that causes the pipeline to stall because of some conflict in the pipe.

Hazards prevent the next instruction in pipe from executing in its turn.

Stalls: The period in which any stages of a pipeline are idle.

They are also referred to as **bubbles** in the pipeline

Types of hazards

Structural Hazards : contention for same hardware resource

Data Hazards : dependency on earlier instruction for the correct sequencing of register reads and writes

Control Hazards : branch/jump instructions stall the pipe until get correct target address into PC



Pipeline conflicts

Three types of hazards are possible in pipeline systems

Structural hazards (Resource conflicts) In a situation when two instructions require the use of the same hardware resource at the same time.

The most common case in which this hazard may arise is in access to memory.

Data Hazards (Data dependency conflicts)

A condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline.

As a result some operation has to be delayed, and the pipeline stalls.

Control /Instruction Hazards (Branch/ Instruction interrupts/Cache miss)

A pipeline be stalled because of a delay in the availability of an instruction

Arise from branch /jump and other instructions that change the value of PC

These instructions stall the pipe until they get correct target address into PC .



Structural Hazards-Solutions

- Pipeline stall (insert bubble)
- Have 2 memory ports for shared instruction-data cache-memory (expensive)
- Have separate instruction cache-memory and data cache-memory



Structural Hazards-Solutions

- A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address.
 - A data dependency occurs when an instruction needs data that are not yet available.
 - An address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
- Pipelined computers deal with such conflicts between data dependencies in a variety of ways.
- Three Generic Data Hazards: RAW, WAR and WOW



Data dependency solutions

Hardware interlocks :

Detect conflicts and delay the progression of an instruction through the pipeline until all necessary data is available.

Ex. Delayed load is performed by the compiler, which inserts NOP (no operation) instructions to ensure that data dependencies are satisfied without the need for additional hardware

- This approach maintains the program sequence by using hardware to insert the required delays.
- **Operand forwarding** : In simple words, if the operands of next instruction is depending on the previous instruction result, after execution (EX) of previous instruction, the result will be directly written (copied) to Register location expected in Next instruction.
 - This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.
- **Data Hazards Remedy (SW)-Delayed load** : (Fig.9)
to delay the loading of the conflicting data by inserting NOP instructions.

```

MOVA R1 , R5
ADD R2, R1, R6
ADD R2, R1, R6

```

Data Hazards Remedy - SW

- Software delay (compiler or machine code programming to insert NOPs)


```

MOVA R1, R5
NOP
NOP
ADD R2, R1, R6
NOP
NOP
ADD R3, R1, R2

```

35

Data Hazards Remedy - HW

- Hardware stalls

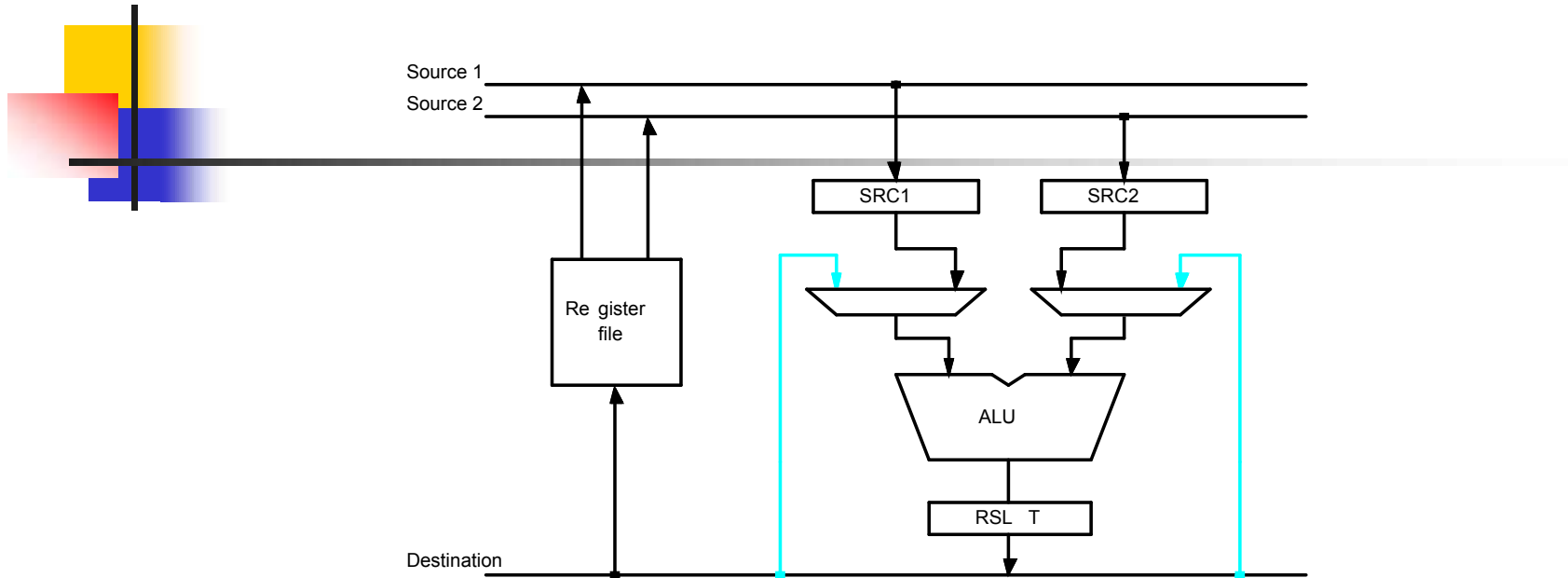

```

MOVA R1, R5
ADD R2 R1 R6
ADD R2 R1, R6

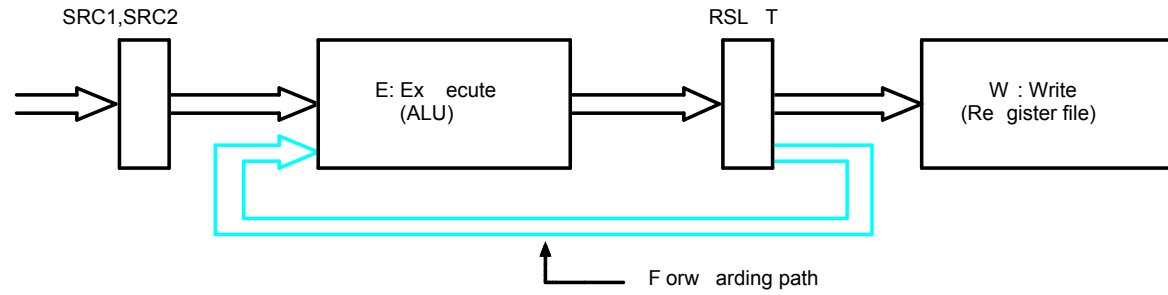
```
- Hardware Data Forwarding
 - Add an extra path connecting ALU outputs to ALU inputs on the next clock

36

Fig.9 Data Forwarding (Reg. Bypassing)



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Figure 9.7. (a) Operand forwarding in a pipelined processor



Handling of branch instructions

- One of the major problems in operating an instruction pipeline is the occurrence of branch instructions.
 - An unconditional branch always alters the sequential program flow by loading the program counter with the target address.
 - In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied.
- Pipelined computers employ following **hardware techniques to minimize the performance degradation** caused by instruction branching.
 - 1) Prefetch target instruction
 - 2) Branch target buffer (BTB)
 - 3) Loop buffer
 - 4) Branch prediction
 - 5) Delayed branch



Handling of branch instructions (cont.)

1) Prefetch target instruction:

- ❖ To prefetch the target instruction in addition to the instruction following the branch.
- ❖ Both are saved until the branch is executed.
- ❖ Cache prefetching is a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in slower memory to a faster local memory before it is actually needed (hence the term 'prefetch'). (Fig.10)

Instruction Queue and Prefetching

Instruction fetch unit

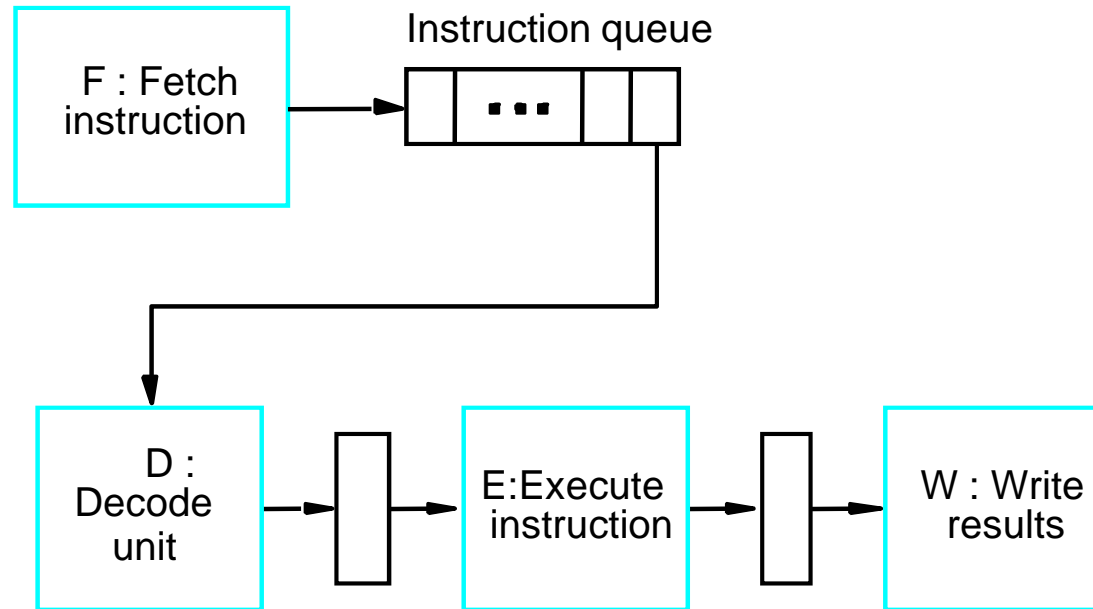


Fig.10 Use of an instruction queue in the hardware organization



Handling of branch instructions

2) Branch target buffer (BTB):

- ❖ The BTB is an associative memory included in the **fetch segment** of the pipeline.
- ❖ Each entry in 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.

Have two pipelines

When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch

- Prefetch each branch into a separate pipeline
- Use appropriate pipeline



Handling of branch instructions

3) Loop buffer:

- ❖ Maintained by fetch stage of pipeline
- ❖ This is a small very high speed register file .
- ❖ Contains the most recently fetched instructions, in sequence.
- ❖ If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer
- ❖ Very good for small loops or iterations
- ❖ If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.



Handling of branch instructions

4) Branch prediction

- ❖ A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- ❖ Reducing the branch penalty requires the branch address to be computed earlier in the pipeline
- ❖ Typically the Fetch unit has dedicated h/w which will identify the branch target address as quick as possible after an instruction is fetched
- ❖ Various techniques can be used to predict whether a branch will be taken or not (Fig.11). The most common techniques are:
 - Static Techniques
 - Dynamic Techniques



Static Techniques:

- a) Predict never taken - continue to fetch sequentially. If the branch is not taken, then there is no wasted fetches.
- b) Predict always taken - fetch from branch target as soon as possible
- c) Predict by opcode - compiler helps by having different opcodes based on likely outcome of the branch

Dynamic Techniques

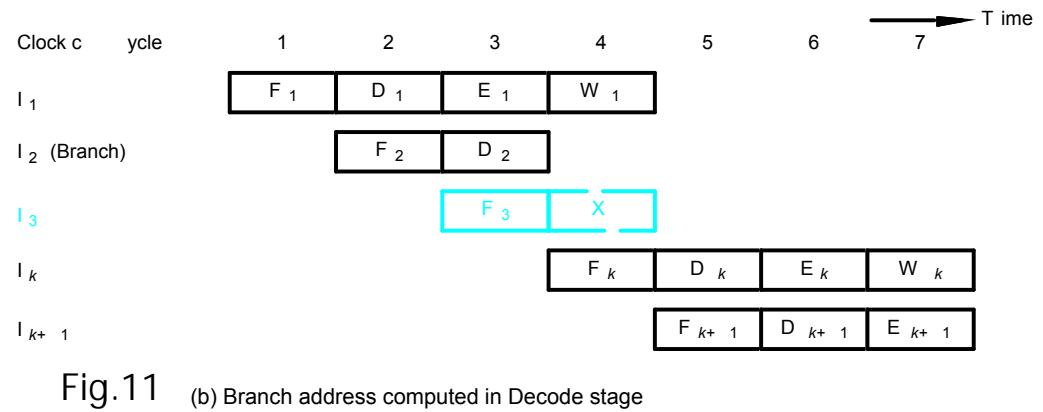
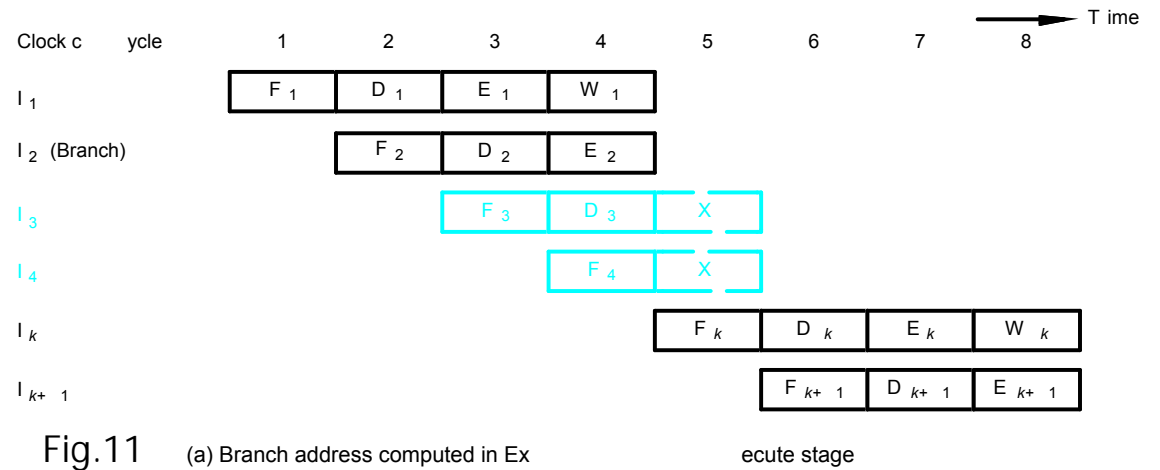
try to improve prediction by recording history of conditional branch

- d) Taken / Not Taken switch - one or more history bits to reflect whether the most recent executions of an instruction were taken or not.
- e) Branch-History Table - small, fully-associative cache to store information about most recently executed branch instructions.

Branch early prediction

- Branch penalty

- Reducing the penalty





Handling of branch instructions

5) Delayed branch:

The compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.

- ❖ A procedure employed in most RISC processors.
- e.g. no-operation instruction



RISC and CISC

CISC : Complex Instruction Set Computer

RISC : Reduced Instruction Set Computer



CISC

CISC instructions sets some common characteristics:

- ❖ A 2-operand format, where instructions have a source and a destination. Register to register, register to memory, and memory to register commands. Multiple addressing modes for memory, including specialized modes for indexing through arrays
- ❖ Variable length instructions where the length often varies according to the addressing mode
- ❖ Instructions which require multiple clock cycles to execute.
- ❖ Easy to program.
- ❖ Makes efficient use of memory



Characteristics of CISC

- ❖ A large of instructions typically from 100 to 250 instructions.
- ❖ A large variety of addressing modes typically from 5 to 200 different modes.
- ❖ Complex and efficient machine instructions.
- ❖ Variable-length instruction formats.
- ❖ Instructions that manipulate operands in memory.
- ❖ Extensive addressing capabilities for memory operations.
- ❖ Relatively few registers.



CISC Disadvantages

- Instruction set & chip hardware become more complex with each generation of computers.
- Many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length - this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.



What is RISC ?

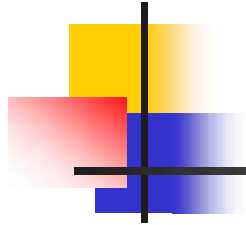
RISC architecture utilizes a small, highly-optimized set of instructions instead of specialized set of instructions often found CISC.

- Characteristic of most RISC processors:
 - **one cycle execution time**: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called PIPELINING
 - **pipelining**: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
 - **large number of registers**: the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory
 - Fixed-length, easily decoded instructions
 - Intelligent compiler



RISC Characteristics

- ❖ Reduced instruction set.
- ❖ Relatively few instructions.
- ❖ Relatively few addressing modes.
- ❖ Less complex, simple instructions.
- ❖ All operations done within the registers of the CPU.
- ❖ Small number of suboperations, with each being executed in one clock cycle
- ❖ Efficient Instruction pipeline
 - ❖ Uses simple decoder
- ❖ Hardwired control unit and machine instructions.
- ❖ Few addressing schemes for memory operands with only two basic instructions, LOAD and STORE
- ❖ Memory access limited to LOAD and STORE instructions



RISC Pipeline



Characteristics of RISC

To use an efficient instruction pipeline

- Suboperations to be executed in one clock cycle.
- Use of fixed-length instruction format to make decoding time same as the register selection.
- RISC instruction pipeline can be **implemented with 2 and 3 segments**.
 - One segment fetches the instruction from program memory
 - The other segment executes the instruction in the ALU
 - Third segment may be used to store the result of the ALU operation in a destination register



Characteristics of RISC (cont.)

The data transfer instructions in RISC are limited to load and store instructions.

- These instructions use register indirect addressing. They usually need three or four stages in the pipeline.
- To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most **RISC machines use two separate buses with two memories.**
- Cache memory: operate at the same speed as the CPU clock
- Ability to execute instructions at the rate of one per clock cycle.
 - In effect, it is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.
 - RISC can achieve pipeline segments, requiring just one clock cycle.



Characteristics of RISC (cont.)

RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems / difficulties associated with data conflicts and branch penalties.



3 Segment Instruction RISC Pipeline

There are three types of instructions:

- ❖ The data manipulation instructions: operate on data in processor registers
- ❖ The data transfer instructions:
- ❖ The program control instructions:



Hardware operation

The **control section** fetches the instruction from program memory into an instruction register.

- The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected.
- The processor unit consists of a number of registers and an arithmetic logic unit (ALU).
- A data memory is used to **load** or **store** the data from a selected register in the register file.
- The instruction cycle can be divided into three sub operations and implemented in three segments



RISC pipeline organization

- I: Instruction fetch
 - Fetches the instruction from program memory
- A: ALU operation
 - The instruction is decoded and an ALU operation is performed.
 - It performs an operation for a data manipulation instruction.
 - It evaluates the effective address for a load or store instruction.
 - It calculates the branch address for a program control instruction.
- E: Execute instruction
 - Directs the output of the ALU to one of three destinations, depending on the decoded instruction.
 - It transfers the result of the ALU operation into a destination register in the register file.
 - It transfers the effective address to a data memory for loading or storing.
 - It transfers the branch address to the program counter.



RISC Pipelining Basics

1. ALU operation with register input and output

two phases of execution for register based instructions

- I: Instruction fetch
- E: Execute

2. Register to memory or memory to register operation

For load and store there will be three phases

- I: Instruction fetch
- E: Execute
 - Calculate memory address
- D: Memory



RISC -Three-Stage Instruction Pipeline

1. Data Manipulation Instructions

I: Instruction Fetch

A: Decode, Read Registers, ALU Operations

E: Write a Register

2. Load and Store Instructions

I: Instruction Fetch

A: Decode, Evaluate Effective Address

E: Register-to-Memory or Memory-to-Register

3. Program Control Instructions

I: Instruction Fetch

A: Decode, Evaluate Branch Address

E: Write Register(PC)



Optimization of RISC Pipelining

Delayed branch

- Leverages branch that does not take effect until after execution of following instruction



Delayed Load

- Consider the operation of the following four instructions:
 1. LOAD: $R1 \leftarrow M[\text{address } 1]$
 2. LOAD: $R2 \leftarrow M[\text{address } 2]$
 3. ADD: $R3 \leftarrow R1 + R2$
 4. STORE: $M[\text{address } 3] \leftarrow R3$
- There will be a **data conflict** in instruction 3 because the operand in R2 is not yet available in the A segment.
- This can be seen from the timing of the pipeline shown in Fig. 12(a).
 - The E segment in clock cycle 4 is in a process of placing the memory data into R2.
 - The A segment in clock cycle 4 is using the data from R2.
- **compiler** has to make sure that the instruction following the load instruction uses the data fetched from memory.



Delayed Load(cont.)

- This concept of delaying the use of the data loaded from memory is referred to as **delayed load**.
- Fig. 12(b) shows the same program with a no-op instruction inserted after the load to R2 instruction.
- Thus the **no-op instruction** is used to advance one clock cycle in order to compensate for the **data conflict** in the pipeline.
- The advantage of the delayed load approach is that the data dependency is taken care of by the **compiler rather than the hardware**.

3-segment pipeline time-space diagram

Clock cycles:	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

12(a) Pipeline timing with data conflict

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

	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

12 (b) Pipeline timing with delayed load



Delayed Branch

- The method used in most RISC processors is to rely on the **compiler to redefine the branches** so that they take effect at the proper time in the pipeline. This method is referred to as **delayed branch**.
- The compiler is designed to analyze the instructions **before and after the branch** and **rearrange the program sequence** by inserting useful instructions in the delay steps.
- It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert **no-op** instructions.



An Example of Delayed Branch

■ Ex. The program of delayed branch:

Load from memory to R1
Increment R2
Add R3 to R4
Subtract R5 from R6
Branch to address X

- In Fig.13(a) the compiler inserts **two no-op instructions** after the branch.
 - The branch address X is transferred to PC in clock cycle 7.
- The program in Fig.13(b) is rearranged by placing the add and subtract instructions **after the branch instruction**.
 - PC is updated to the value of X in clock cycle 5.

Example: delayed branch

Load from memory to R1
 Increment R2
 Add R3 to R4
 Subtract R5 from R6
 Branch to address X

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. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

13 (a) Using no-operation instructions

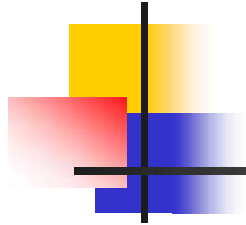
Example: delayed branch(cont.)

Load from memory to R1
 Increment R2
 Add R3 to R4
 Subtract R5 from R6
 Branch to address X

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. Instruction in X						I	A	E

13 (b) Rearranging the instructions

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



Vector Processing



Introduction

- In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.
- Computers with vector processing capabilities are in demand in specialized applications. e.g.
 - Long-range weather forecasting
 - Petroleum explorations
 - Seismic data analysis
 - Medical diagnosis
 - Artificial intelligence and expert systems
 - Image processing
 - Mapping the human genome



Introduction(cont.)

To achieve the required level of high performance it is necessary

- ❖ to utilize the **fastest and most reliable hardware**
- ❖ apply innovative procedures
(**vector and parallel processing techniques**)



Vector Operations

- Many scientific problems require arithmetic operations on large arrays of numbers.
- A vector is an ordered set of a 1-D array of data items.
- A vector V of length n is represented as a row vector by $V = [v_1, v_2, \dots, v_n]$.
- To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

```
DO 20 I = 1, 100  
20  C(I) = B(I) + A(I)
```



Vector Operations(cont.)

- This is implemented in machine language by the following sequence of operations.

```
Initialize I=0  
20 Read A(I)  
Read B(I)  
Store C(I) = A(I)+B(I)  
Increment I = I + 1  
If I 100 go to 20  
Continue
```

- A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop.

$$C(1:100) = A(1:100) + B(1:100)$$



Vector Operations (cont.)

- A possible instruction format for a vector instruction is shown in Fig.14.
 - This assumes that the vector operands reside in **memory**.
- It is also possible to design the processor with a large number of **registers** and store all operands in registers prior to the addition operation.
 - The base address and length in the vector instruction specify a group of CPU registers.



Vector Processor

Instruction format

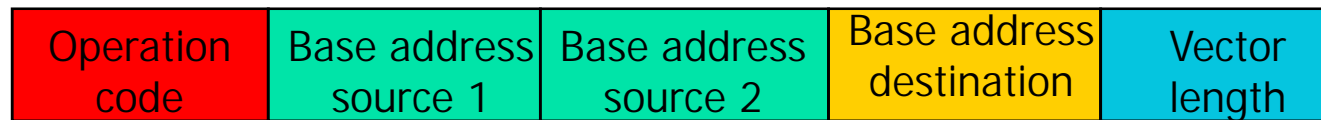


Fig.14.



Matrix Multiplication

- The multiplication of two $n \times n$ matrices consists of n^2 inner products or n^3 multiply-add operations.
 - Consider, for example, the multiplication of two 3×3 matrices A and B.
 - $C = A * B$ and $C = C_{11} + C_{22} + C_{33}$
 - $C_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$; $C_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$
 - $C_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$
 - This requires 3 multiplication and (after initializing c_{11} to 0) 3 additions.
- In general, the inner product consists of the sum of k product terms of the form $C = A_1B_1 + A_2B_2 + A_3B_3 + \dots + A_kB_k$.
 - In a typical application k may be equal to 100 or even 1000.
- The inner product calculation on a pipeline vector processor is shown in Fig.12.

Pipeline for calculating an inner product

$$\begin{aligned} C &= A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \dots \\ &+ A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \dots \\ &+ A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \dots \\ &+ A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \dots \end{aligned}$$

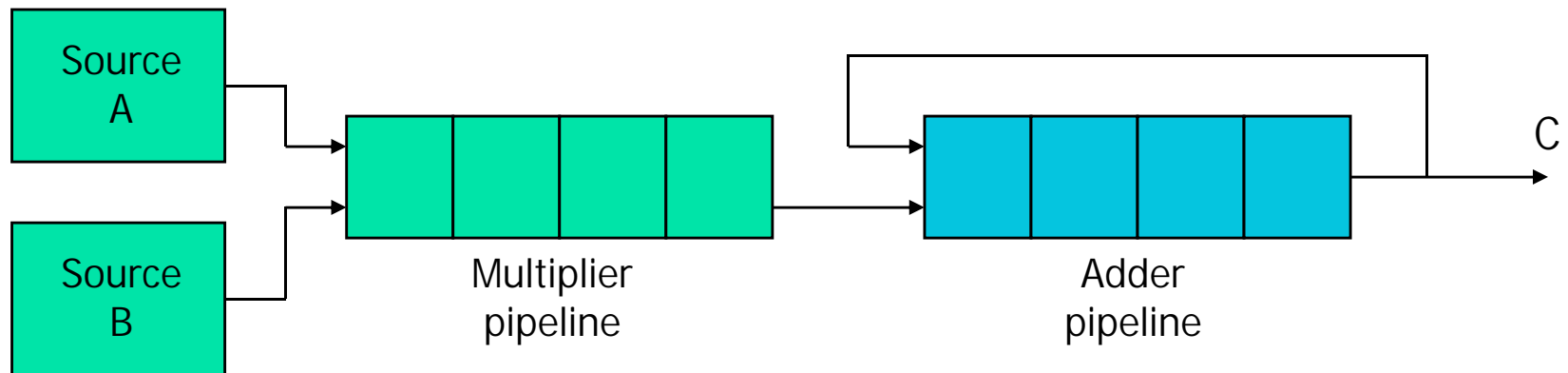


Fig.12

Pipeline for calculating an inner product

$$\begin{aligned} C &= A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \dots \\ &+ A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \dots \\ &+ A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \dots \quad \text{Product } p_i = A_i B_i \\ &+ A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \dots \end{aligned}$$

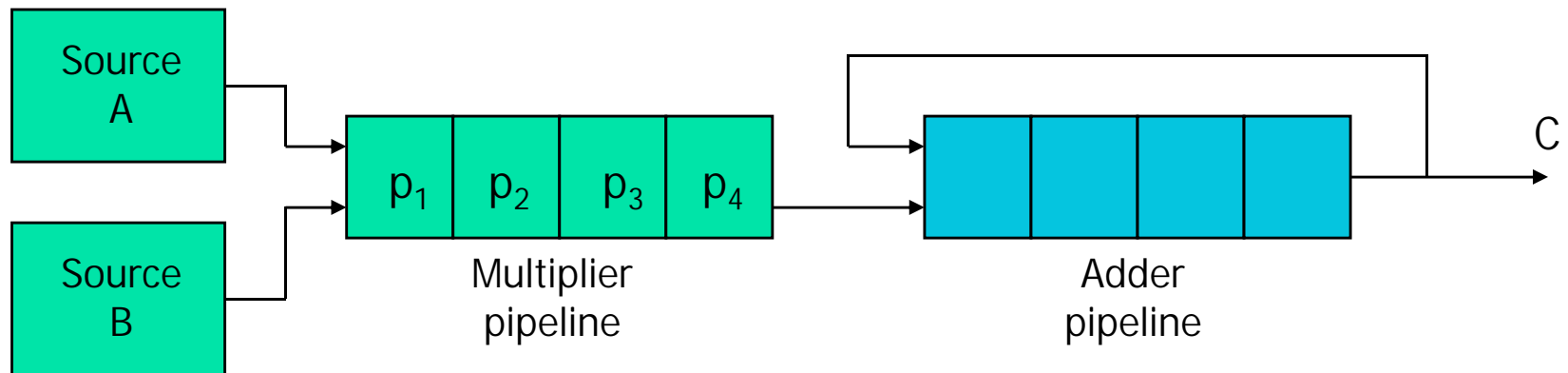


Fig.12 (a)-After 4 units of time

Pipeline for calculating an inner product

$$\begin{aligned}
 C = & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \dots \\
 & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \dots \\
 & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \dots \text{ Product } p_i = A_i B_i \\
 & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \dots
 \end{aligned}$$

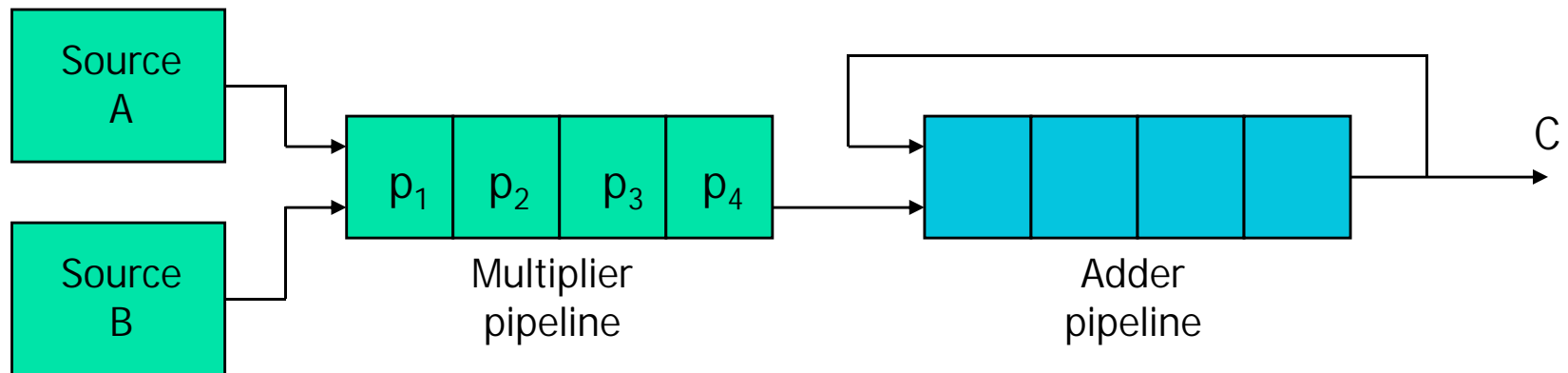


Fig.12 (a)-After 8 units of time

Pipeline for calculating an inner product

$$\begin{aligned} C = & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \dots \\ & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \dots \\ & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \dots \text{ Product } p_i = A_i B_i \\ & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \dots \end{aligned}$$

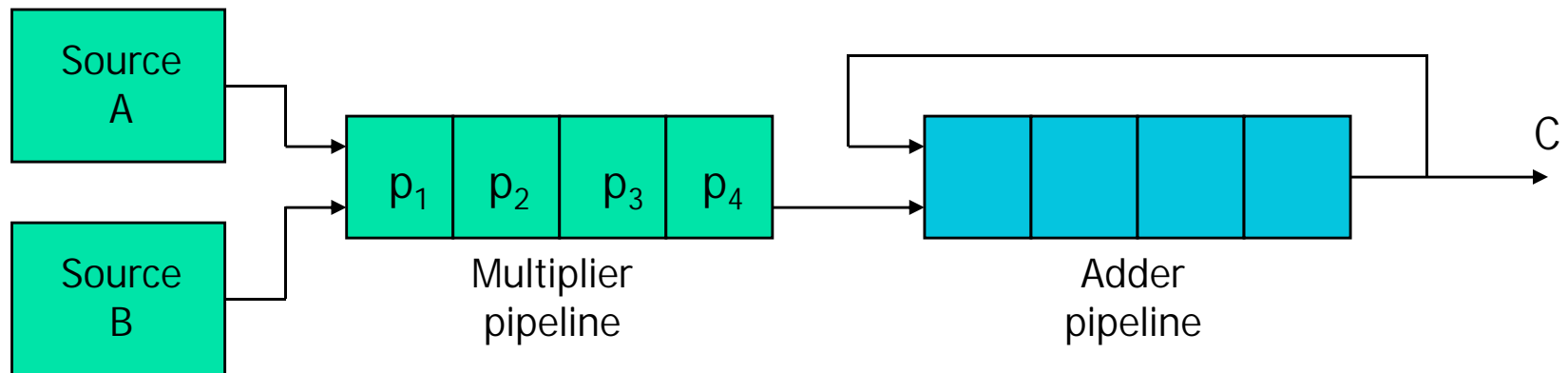


Fig.12 (a)-After 4 units of time

Pipeline for calculating an inner product

$$\begin{aligned} C &= A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \dots \\ &+ A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \dots \\ &+ A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \dots \quad \text{Product } p_i = A_i B_i \\ &+ A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \dots \end{aligned}$$

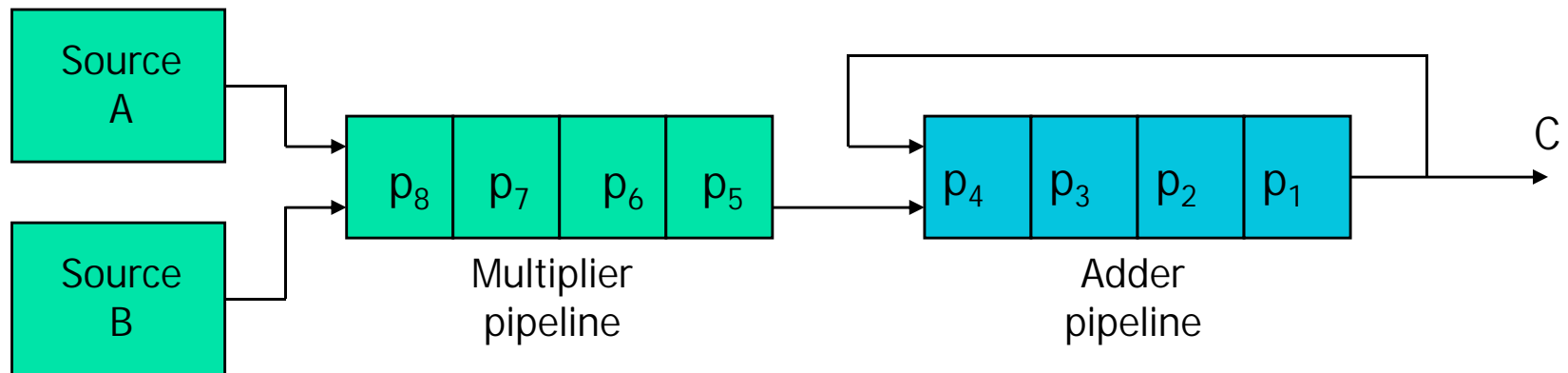


Fig.12 (a)-After 8 units of time

Pipeline for calculating an inner product

Product $p_i = A_i B_i$

$$\begin{aligned}
 C &= A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \dots \\
 &+ A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \dots \\
 &+ A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \dots \\
 &+ A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \dots
 \end{aligned}$$

$$\begin{aligned}
 C &= p_1 + p_5 + p_9 + p_{13} \\
 &+ p_2 + p_6 + p_{10} + p_{14} \\
 &+ p_3 + p_7 + p_{11} + p_{15} \\
 &+ p_4 + p_8 + p_{12} + p_{16}
 \end{aligned}$$

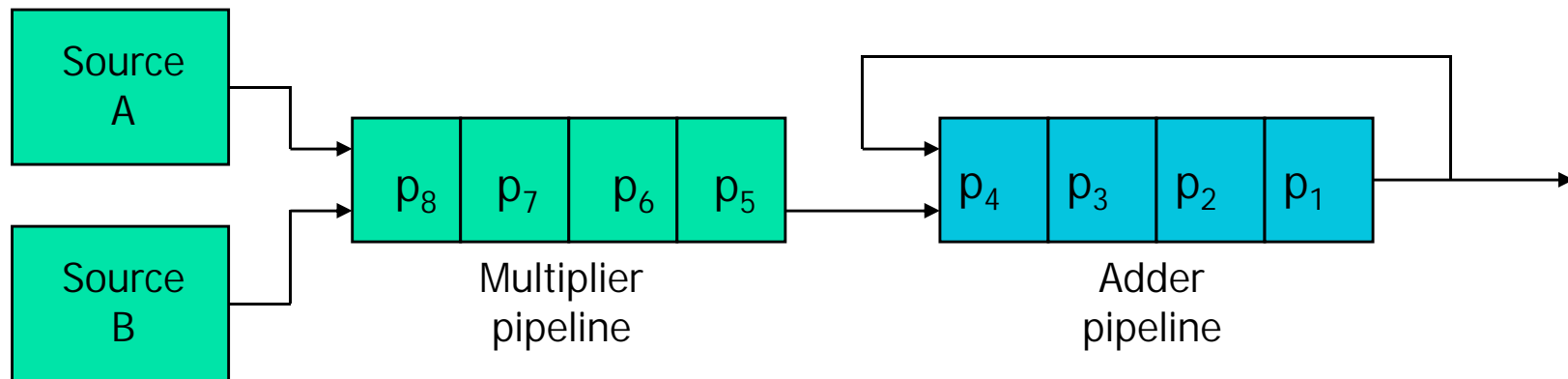


Fig.12 (a)-After 9 units of time $C = p_1 + p_5$

and so on



Memory Interleaving

- **Pipeline** and **vector processors** often require simultaneous access to memory from two or more sources.
 - An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.
 - An arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time.
- Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses.
- The advantage of a modular memory is that it allows the use of a technique called interleaving.
 - A memory module is a memory array together with its own address and data registers.
- Fig.13 shows a memory unit with four modules.

Memory Organization-Multiple modules

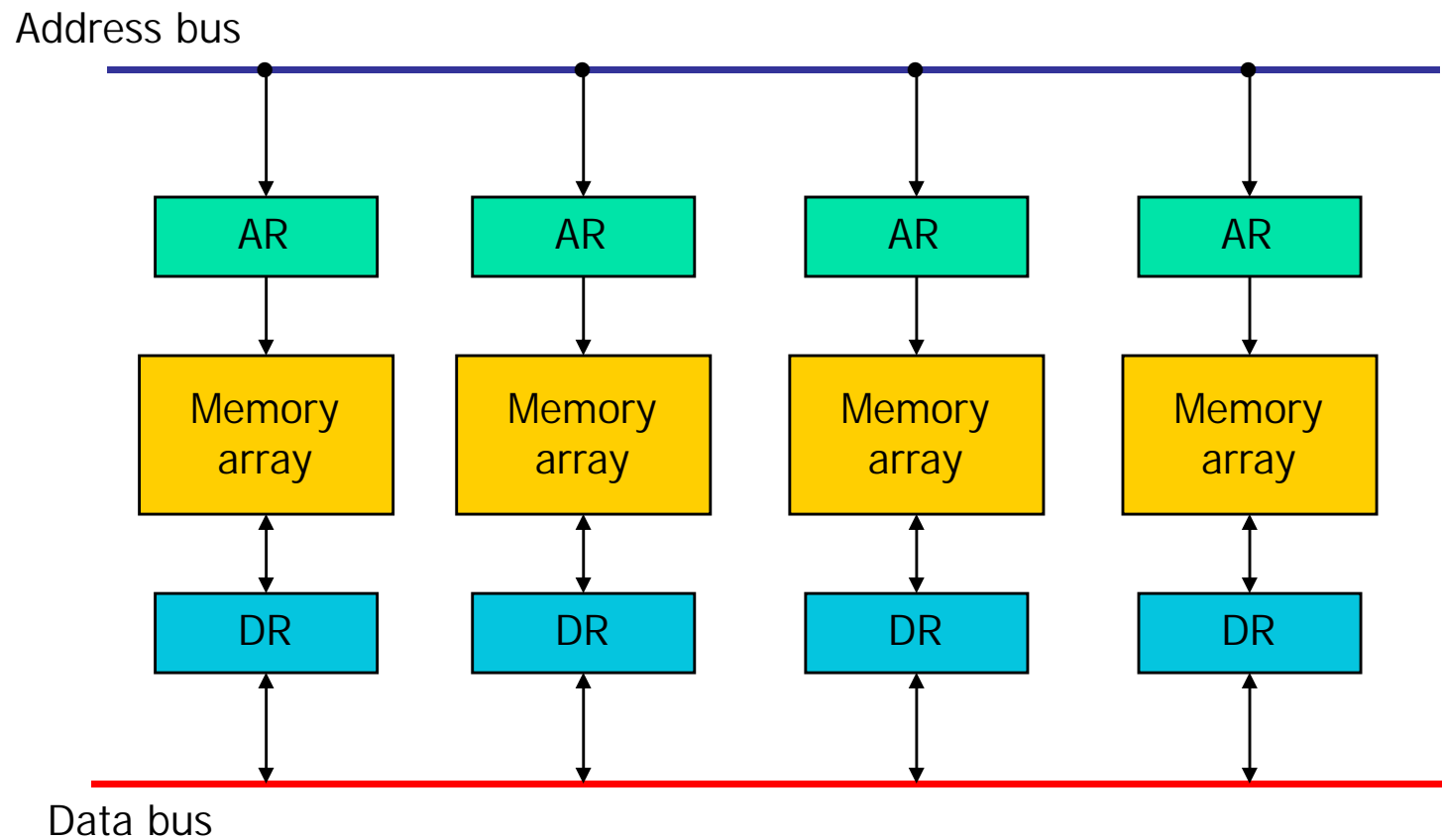


Fig.13



Memory Interleaving(cont.)

- Memory interleaving is the technique used to increase the throughput.
- The core idea is to split the memory system into independent banks, which can answer read or write requests independently in parallel.
- In an interleaved memory, different sets of addresses are assigned to different memory modules.
- By staggering the memory access, the effective memory cycle time can be **reduced by a factor close to the number of modules**.
- The ultimate purpose is to match the memory bandwidth with the bus bandwidth and with the processor bandwidth.
- There are two-address format for memory interleaving the address space.

Low order interleaving and High order interleaving



Lower order interleaving (LOI)

- LOI spreads contiguous memory location across the modules horizontally.

- This implies that the low order bits of the memory address are used to identify the memory module.
- High order bits are the word addresses (displacement) within each module .

High order interleaving

- HOI uses the high order bits as the module address or to identify the memory module.
- Low order bits are the word addresses within each module.

Where is it implemented ?

- Implemented on main memory , Which is slow as compared to Cache. And Main memory having less bandwidth.

Memory bandwidth

- Is the rate at which data can be read and write into a memory by a processor. Memory bandwidth is usually expressed in units of bytes/sec.



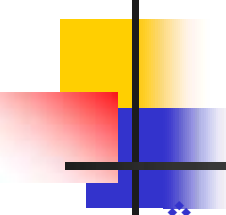
Modules and addresses in Main Memory

- ❖ The main memory is built with multiple modules (chips).
- ❖ These memory modules are connected to a bus and other resources (Processor, I/O) are also connected memory module having memory addresses.
- ❖ Each memory module return one word per cycle.
- ❖ It is possible to present different addresses to different memory modules.
- ❖ So the Parallel access of multiple word can be done concurrently (one cycle).
- ❖ This called parallel access in term of pipelined fashion.
- ❖ Main memory is often block-accessed at consecutive addresses.
- ❖ Block access is needed for fetching a sequence of instructions or for accessing a linearly ordered data structure.
- ❖ Each block access may correspond to the size of a cache block.
- ❖ Therefore it is desirable to design the memory to facilitate the block access of contiguous words.



Supercomputers

- A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a **supercomputer**.
 - To speed up the operation, the components are packed tightly together to minimize the distance that the electronic signals have to travel.
- This is augmented by instructions that process vectors and combinations of scalars and vectors.
- A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing.
 - It is equipped with **multiple functional units** and each unit has its own **pipeline** configuration.
- It is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.

- 
-
- ❖ A Main memory formed with $m = 2^a$ memory modules.
 - ❖ With each module contains $w = 2^b$ words of memory cells.

Low order interleaving

- ❖ lower order '**a**' bits are used to identify the memory module.
- ❖ higher order '**b**' bits are the word addresses (displacement) within each module.

High order interleaving

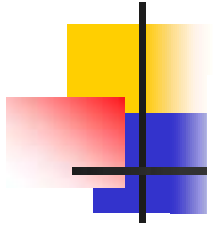
- ❖ higher order '**a**' bits are used as the module address
- ❖ low order '**b**' bits are the word address in each module.

Low order interleaving

4 Way interleaved Memory

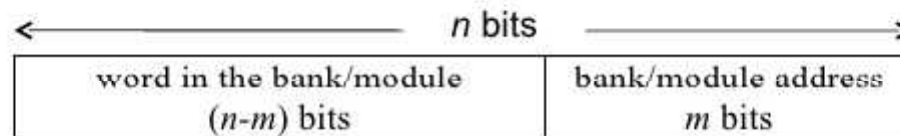
Low order interleaving

	Chip 00	Chip 01	Chip 10	Chip 11			
C[0]	M[0]	C[0]	M[1]	C[0]	M[2]	C[0]	M[3]
C[1]	M[4]	C[1]	M[5]	C[1]	M[6]	C[1]	M[7]
C[1023]	M[4092]	C[1023]	M[4093]	C[1023]	M[4094]	C[1023]	M[4095]



Low-Order Interleaving (LOI)

Address Format



Module 0

Module 1

Module 2

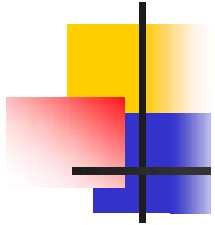
Module 3

0
4
8
12

1
5
9
13

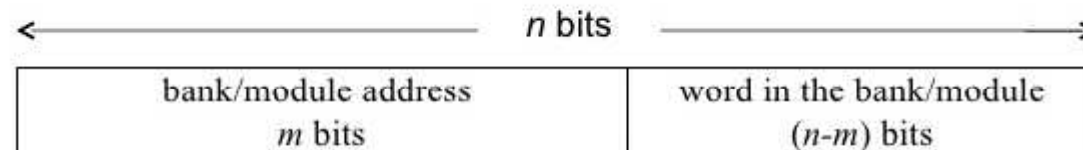
2
6
10
14

3
7
11
15



High-Order Interleaving (HOI)

Address Format



Module 0

0
1
2
3

Module 1

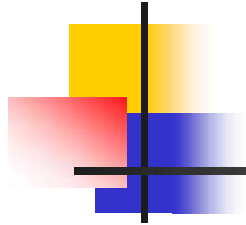
4
5
6
7

Module 2

8
9
10
11

Module 3

12
13
14
15



Array Processors



Introduction

- An array processor is a processor that performs computations on large arrays of data.
- The term is used to refer to two different types of processors.
 - Attached array processor:
 - Is an auxiliary processor.
 - It is intended to improve the performance of the host computer in specific numerical computation tasks.
 - SIMD array processor:
 - Has a single-instruction multiple-data organization.
 - It manipulates vector instructions by means of multiple functional units responding to a common instruction.



Attached Array Processor

Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.

- Parallel processing with multiple functional units
- Fig.14 shows the interconnection of an attached array processor to a host computer.
- The objective of the attached array processor is to provide **vector manipulation capabilities** to a conventional computer at a fraction of the cost of supercomputer.

Attached Array Processor with host computer

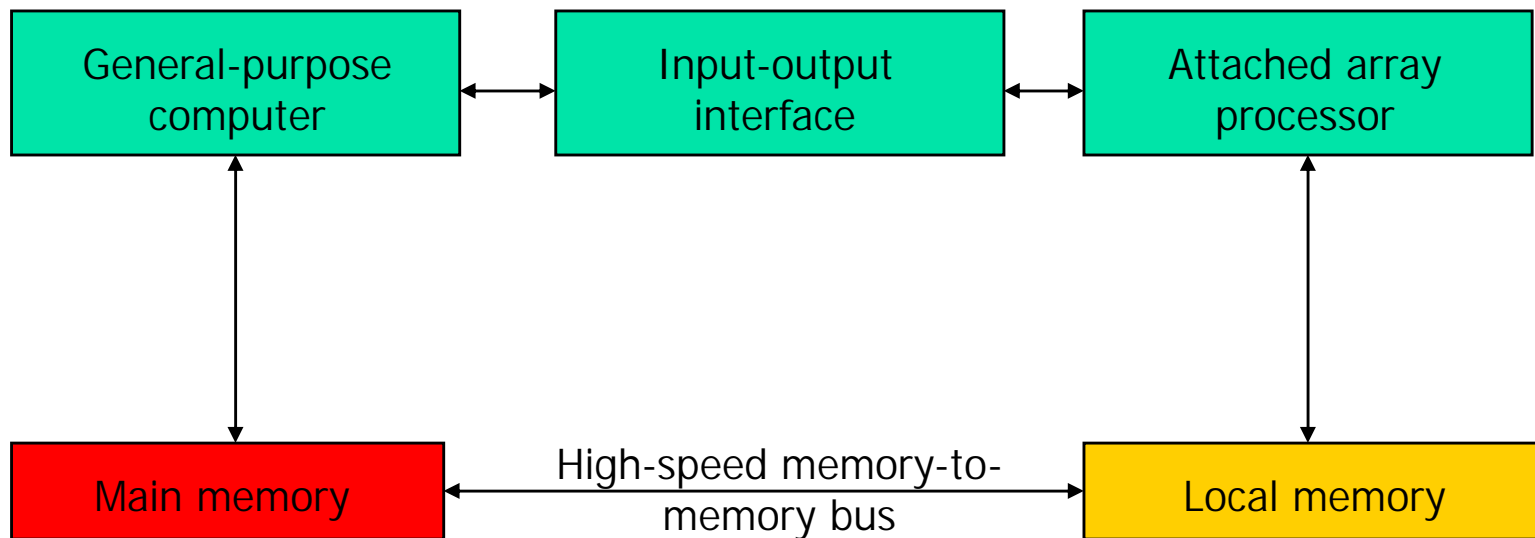


Fig.14.



SIMD Array Processor

- An SIMD array processor is a computer with multiple processing units operating in parallel.
- A general block diagram of an array processor is shown in Fig.15.
 - It contains a set of identical processing elements (PEs), each having a local memory M.
 - Each PE includes an ALU, a floating-point arithmetic unit, and working registers.
 - Vector instructions are broadcast to all PEs simultaneously.
- Masking schemes are used to control the status of each PE during the execution of vector instructions.
 - Each PE has a flag that is set when the PE is active and reset when the PE is inactive.

SIMD Array Processor organization

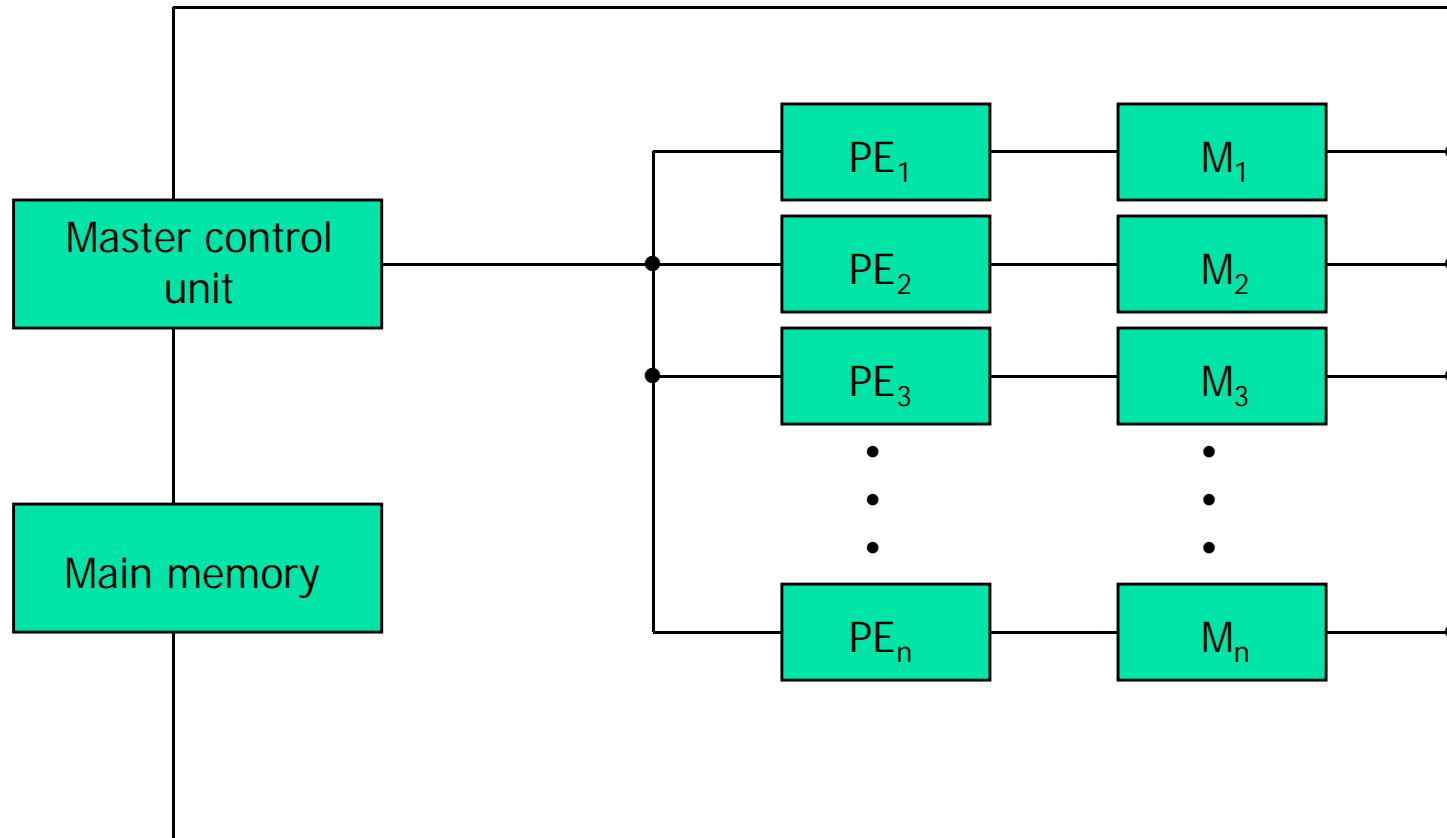


Fig.15



SIMD Array Processor(cont.)

- For example, the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp.
 - Are highly specialized computers.
 - They are suited primarily for numerical problems that can be expressed in vector or matrix form.