UNIT 5

Pipelining: Introduction, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processors, Hazards.

Multiprocessors: Characteristics of Multiprocessors, Interconnection Structures, Inter processor arbitration, Inter processor communication and synchronization, Cache coherence.

Parallel Processing

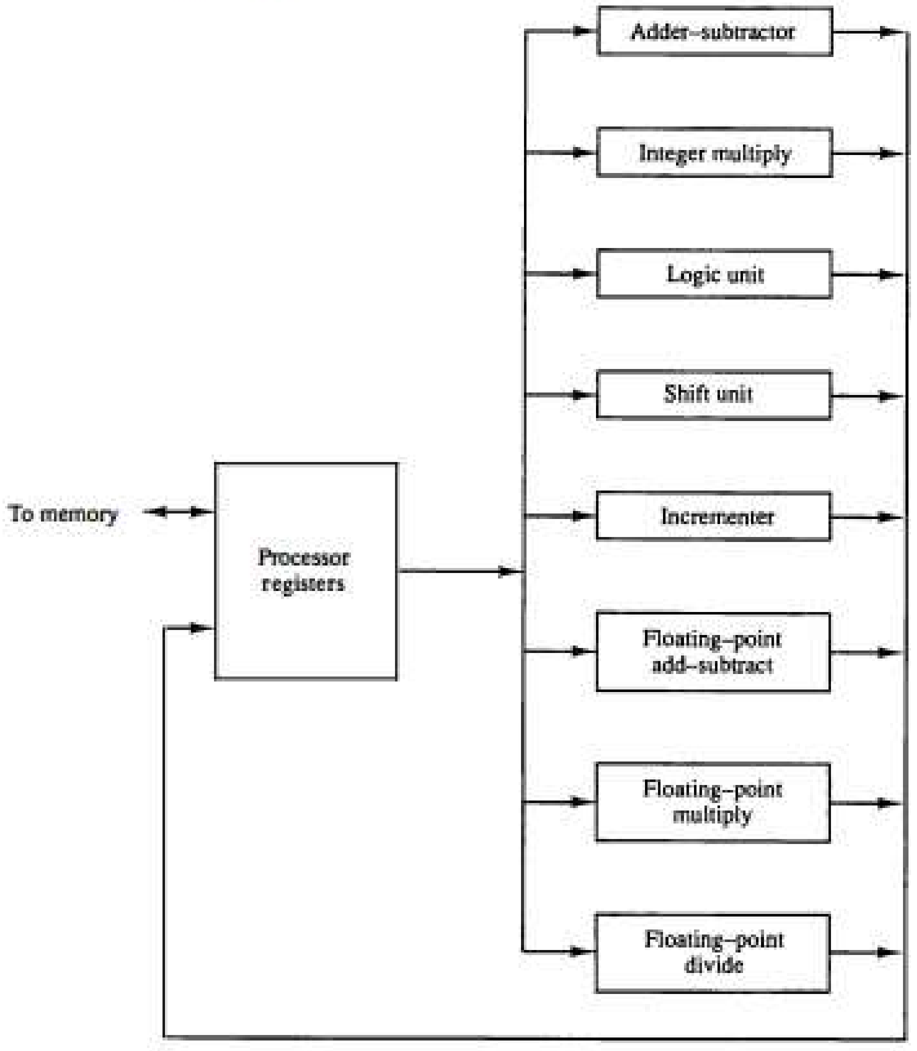
- Parallel Processing
  - *Simultaneous* data processing tasks for the purpose of increasing the computational speed
  - Perform *concurrent* data processing to achieve faster execution time

  - Multiple Functional Unit :

    Parallel Processing Example

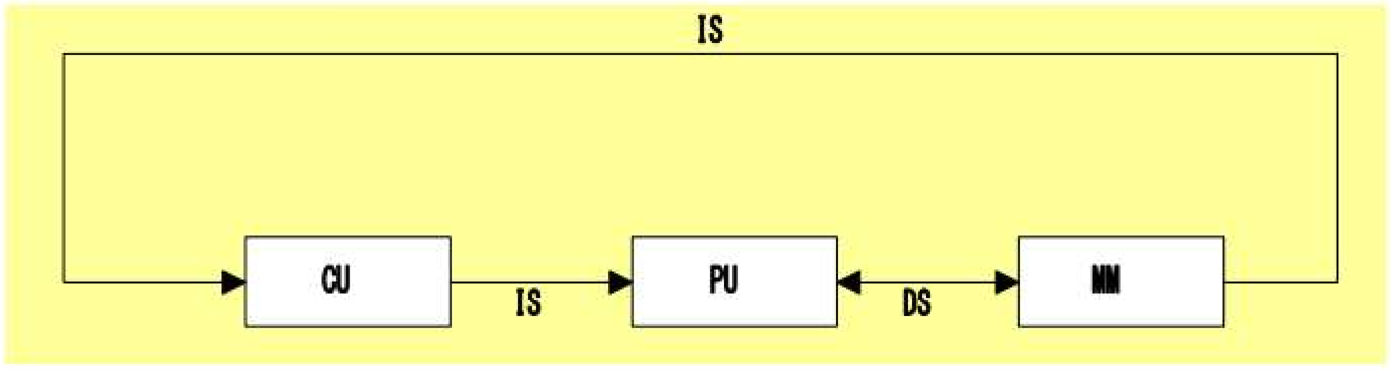
    - ***Separate the execution unit into eight functional units operating in parallel***

Figure 9-1 Processor with multiple functional units.



- Computer Architectural Classification
  - Data-Instruction Stream : Flynn
  - Serial versus Parallel Processing : Feng
  - Parallelism and Pipelining : Handler
- Flynn's Classification
  - 1) **SISD** (Single Instruction stream - Single Data stream)
    - for practical purpose: only one processor is useful
    - Example systems : Amdahl 470V/6, IBM 360/91

- The sequence of instructions read from memory constitutes an instruction stream.
- The operations performed on the data in the processor constitutes a data stream .



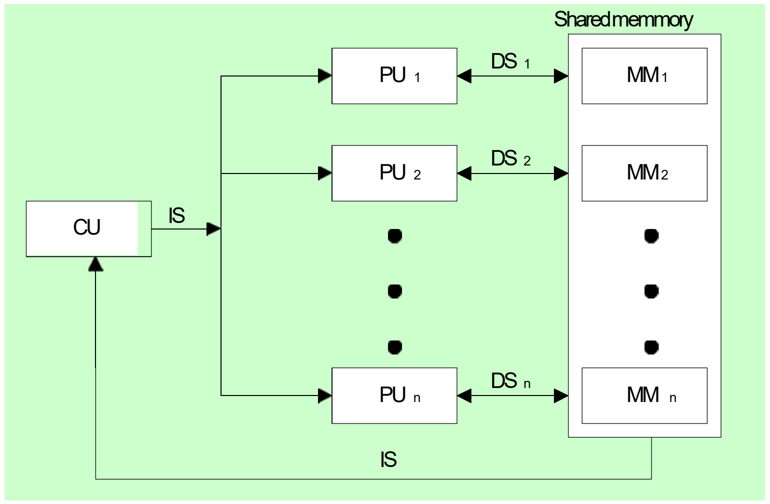
- 2) **SIMD**

  (Single Instruction - Multiple Data stream)

  » vector or array operations

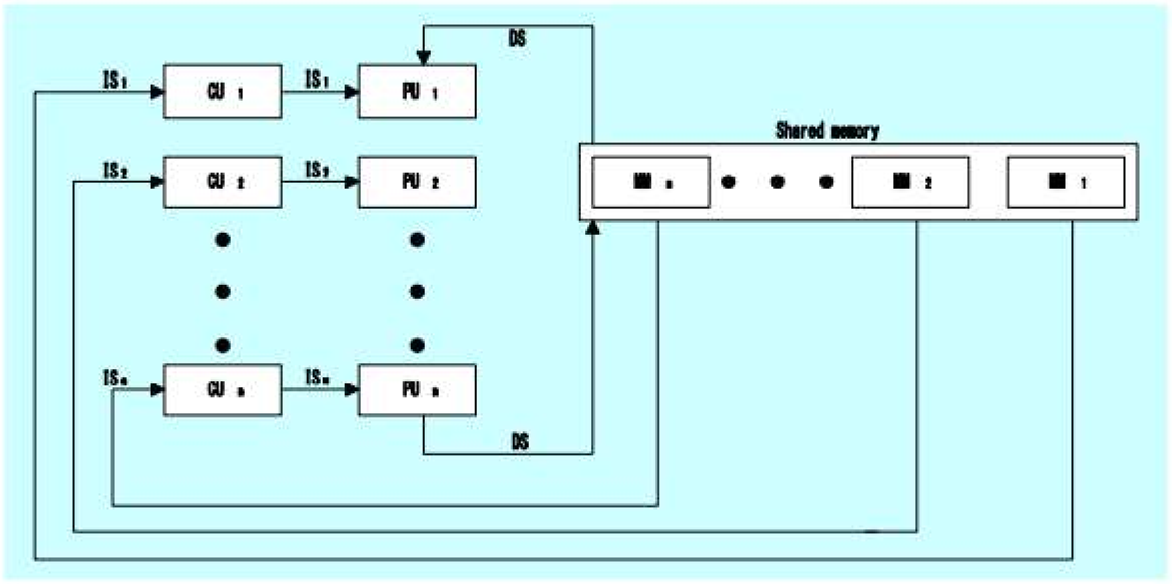
  - one vector operation includes many operations on a data stream

  » Example systems : CRAY -1, ILLIAC-IV

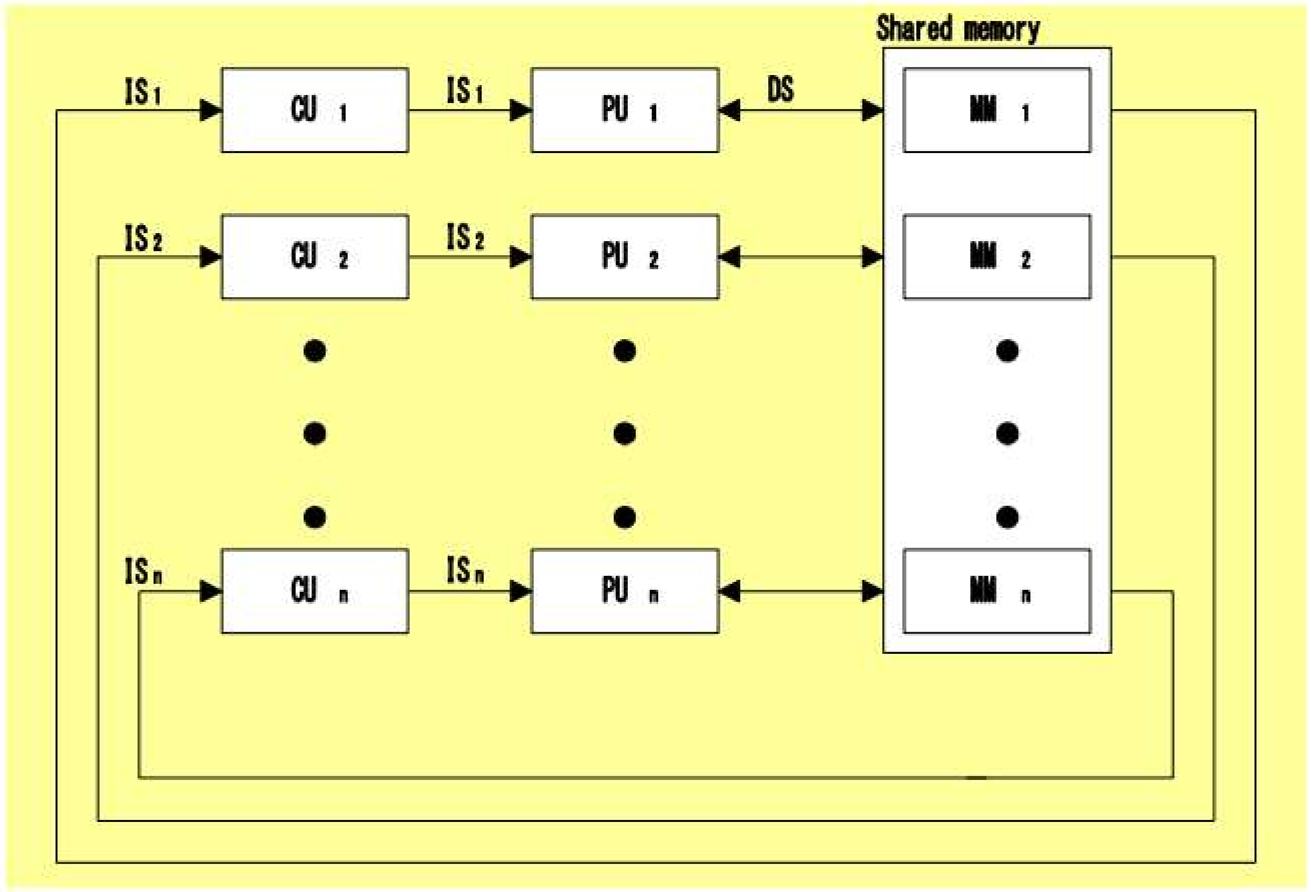


- 3) **MISD**
  (Multiple Instruction - Single Data stream)
  - » Data Stream Bottle neck

- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.



- 4) **MIMD**
  (Multiple Instruction - Multiple Data stream)
  » Multiprocessor 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.

- One type of parallel processing that does not fit Flynn's classification is pipelining.

- Parallel processing computers are required to meet the demands of large scale computations in many scientific, engineering, military, medical, artificial intelligence, and basic research areas.

- The following are some representative applications of parallel processing computers:

- Numerical weather forecasting
- Computational aerodynamics
- Finite-element analysis
- Remote-sensing applications
- Genetic engineering
- Weapon research and defence

Pipelining

- Pipelining
  - Decomposing a sequential process into sub operations
  - Each subprocess is executed in a special dedicated segment concurrently

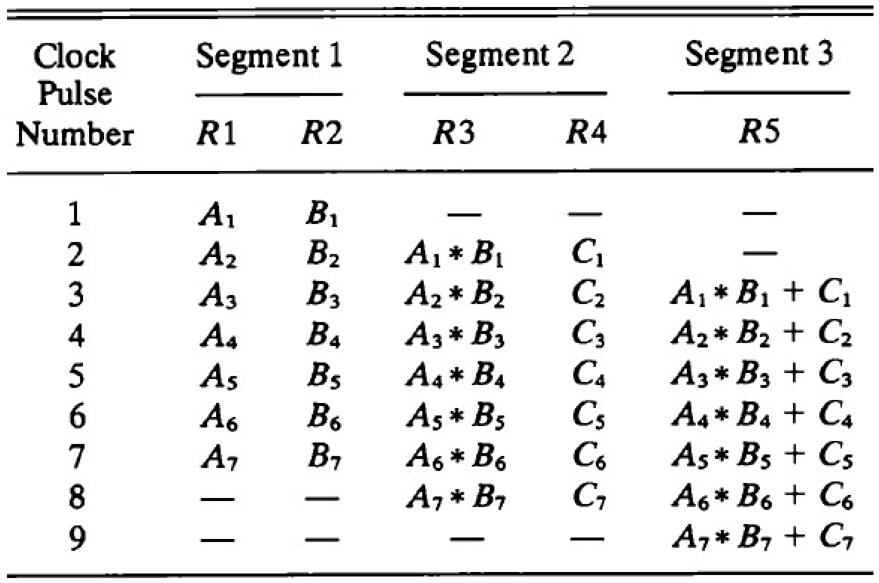
Pipelining Example

- Multiply and add operation : $Ai * Bi + Ci$ (for i = 1, 2, …, 7)
- 3 Suboperation Segment

»1) $R1 \leftarrow Ai,\ R2 \leftarrow Bi$: Input Ai and Bi

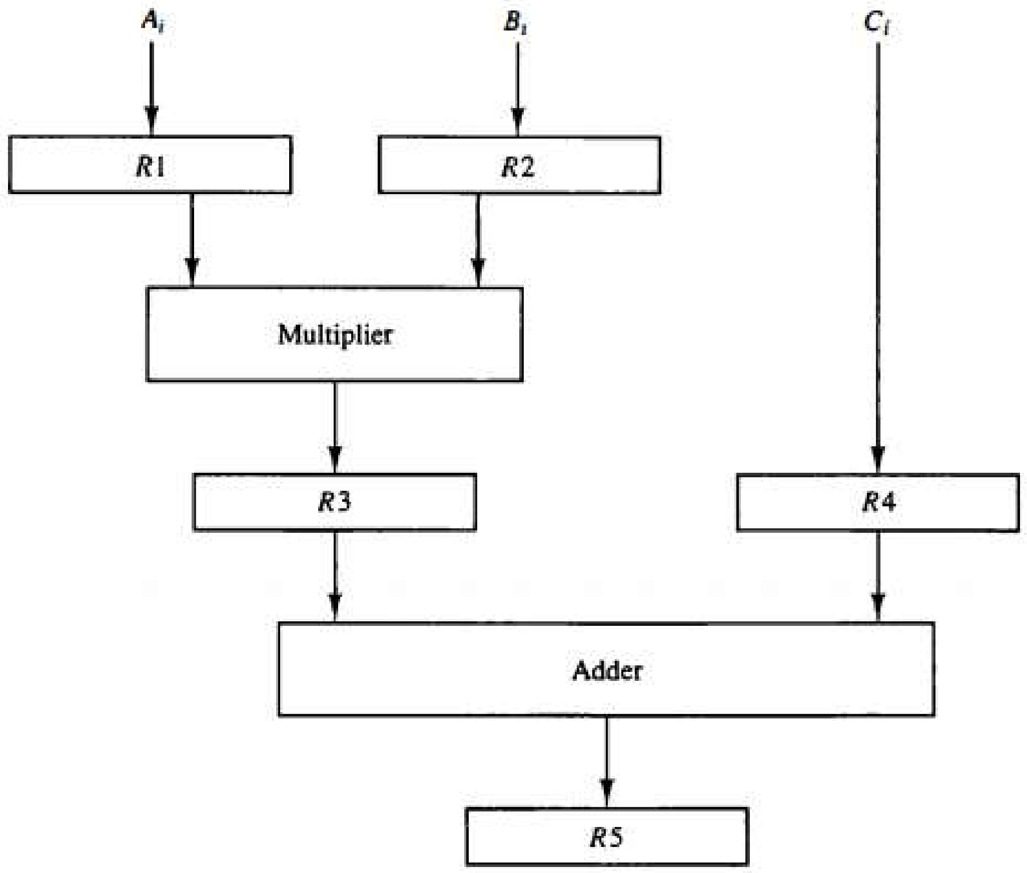
»2) $R3 \leftarrow R1 * R2,\ R4 \leftarrow Ci$: Multiply and input Ci

»3) $R5 \leftarrow R3 + R4$: Add Ci



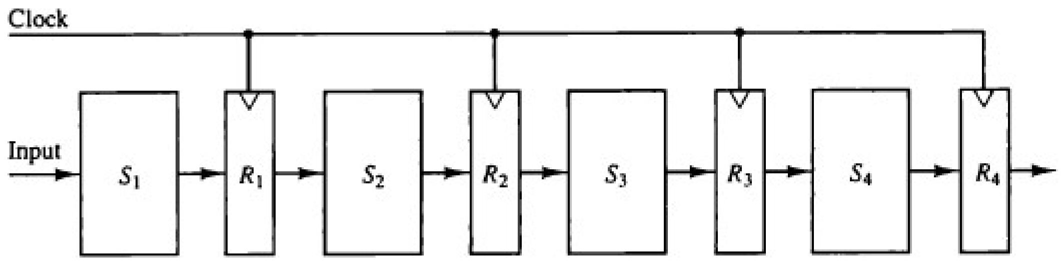
Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	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	—	—	—	—	$A_7 * B_7 + C_7$

Figure 9-2 Example of pipeline processing.



General considerations

- 4 segment pipeline :
  - » **S** : Combinational circuit for Suboperation
  - » **R** : Register(intermediate results between the segments)
- Space-time diagram :
  - » Show segment utilization as a function of time
- Task : T1, T2, T3,…, T6
  - » Total operation performed going through all the segment



Segment \ Clock cycles	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Speedup

- Given a *k*-segment pipeline with a clock cycle of t_p that is used to execute *n* tasks.
  - The first task requires kt_p to complete the operation.
  - The remaining tasks are completed one per clock cycle, requiring an additional $(n-1)t_p$.
  - The total execution time is $(k + n-1)t_p$
- A nonpipelined unit would require nt_n to complete these tasks.
- The speedup is the ratio

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

Theoretical Speedup

- As the tasks increase n is much larger than $k - 1$ and $k + n - 1 \rightarrow n$. Thus the speedup becomes

$$S = t_n / t_p$$

- If we assume that the task takes as long with or without pipelining, we have $t_n = kt_p$, which yields:

$$S = k$$

- Therefore k is the theoretical maximum speedup.

Speedup-An example

Let the time it takes to process a sub operation in each segment be equal to t_p= 20 ns.

Assume that the pipeline has k = 4 segments
n = 100 tasks

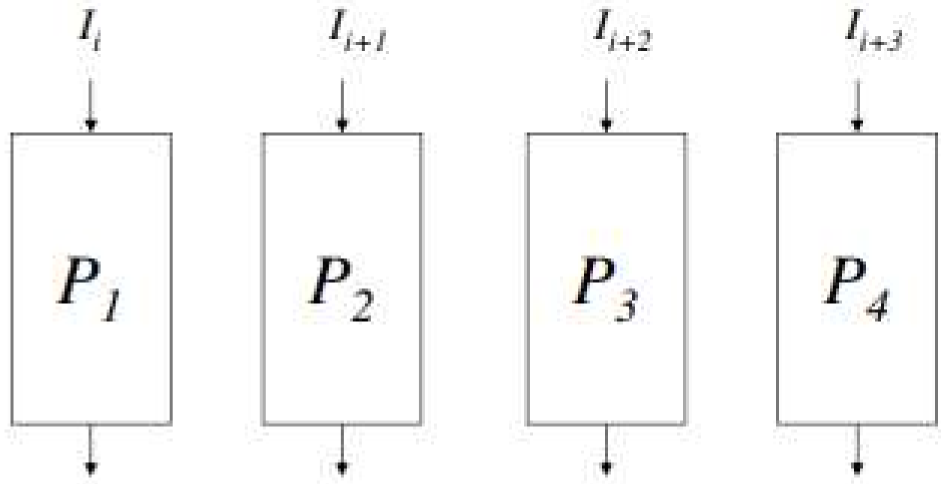
The pipeline system will take $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns

Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a non pipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks.

The speedup ratio =8000/2060 = 3.88.

As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline.

To reach the maximum theoretical speedup, we need to construct multiple functional units in parallel



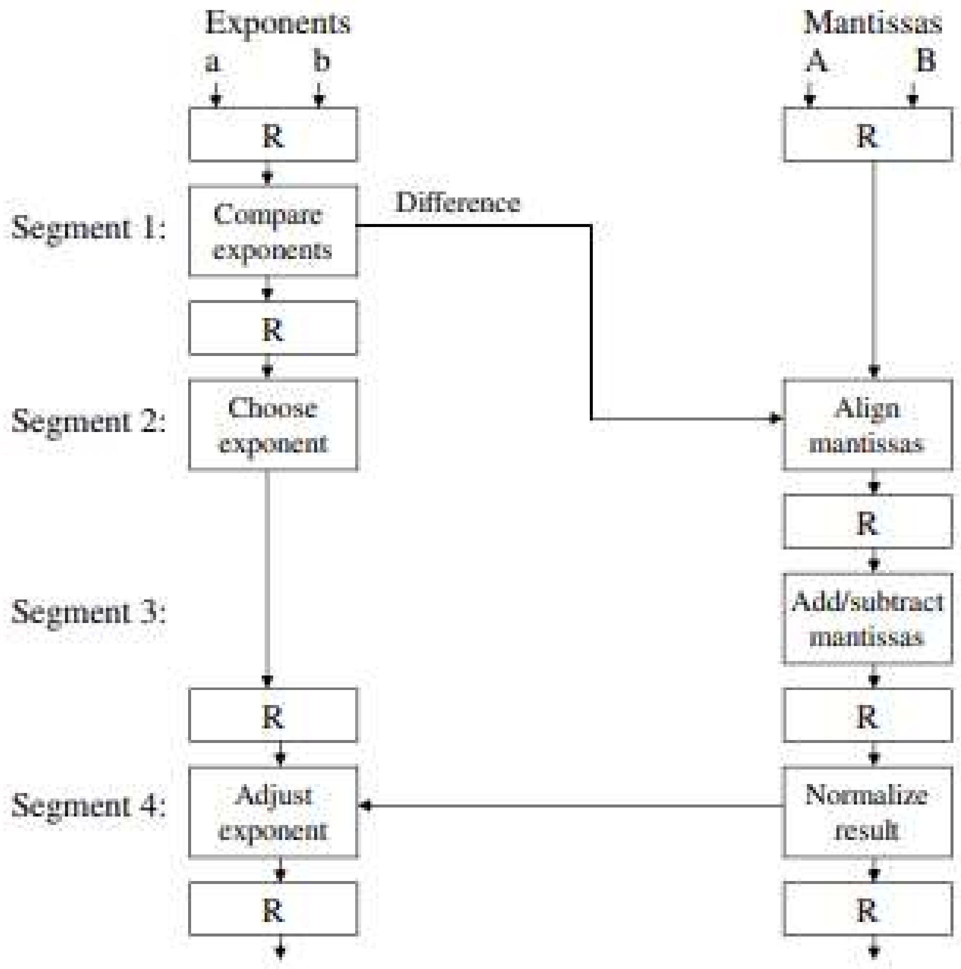
Applying Pipelining

- There are two areas where pipeline organization is applicable:
  - Arithmetic pipelining
    - divides an arithmetic operation into suboperations for execution in the pipeline segments.
  - Instruction pipelining
    - operates on a stream of instructions by operlapping the fetch, decode, and execute phases of the instruction cycles.

Arithmetic Pipelining

- Pipelined arithmetic units are used to implement floating point operations, fixed point multiplication, etc.
- Floating point operations are readily decomposed into suboperations that can be handled separately.

Pipeline for Floating Point Addition/Subtraction



- Arithmetic Pipeline
  - Floating-point Adder Pipeline Example :
  - Add / Subtract two normalized floating-point binary number
    - » $X = A \times 2^a = 0.9504 \times 10^3$
    - » $Y = B \times 2^b = 0.8200 \times 10^2$

4 segments sub operations

1) Compare exponents by subtraction :

$3 - 2 = 1$

$X = 0.9504 \times 10^3$

$Y = 0.8200 \times 10^2$

2) Align mantissas

$X = 0.9504 \times 10^3$

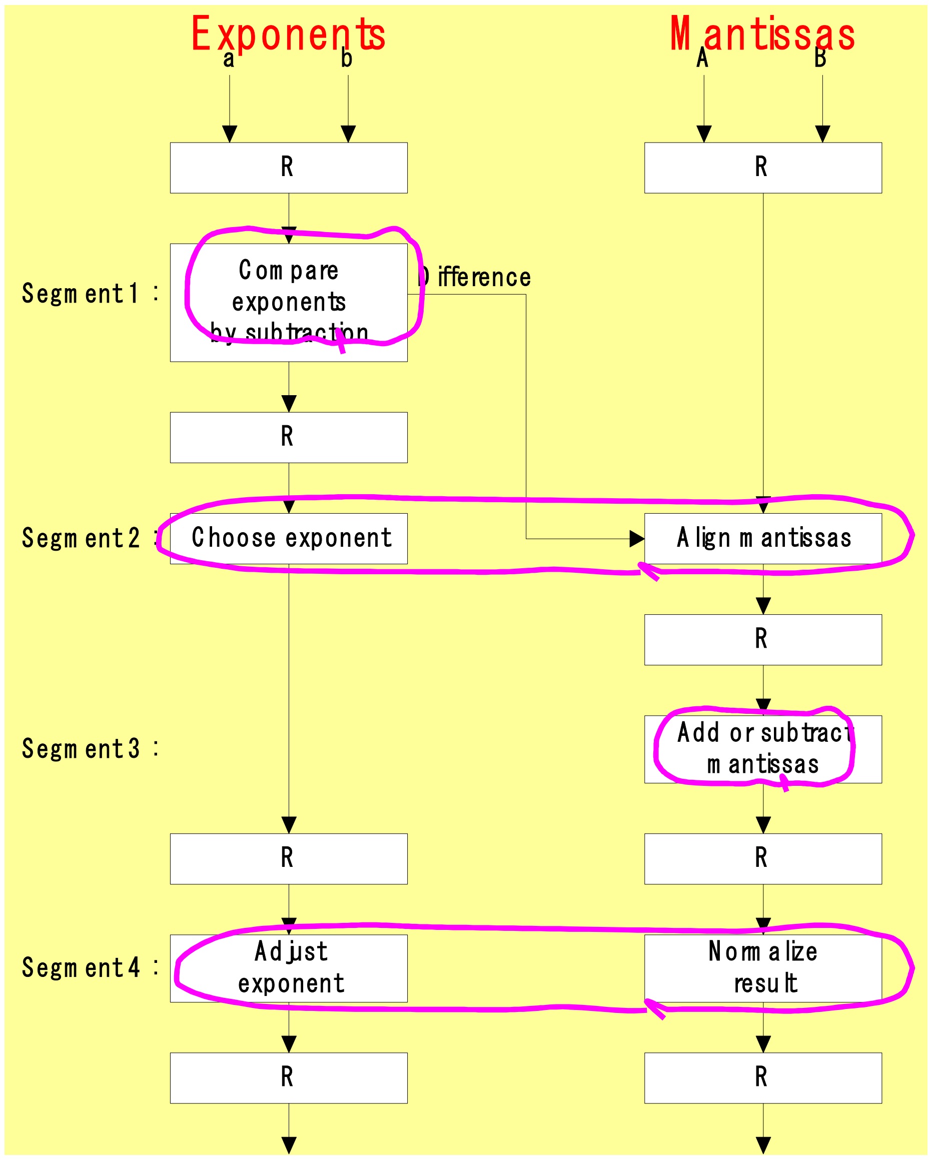
$Y = 0.08200 \times 10^3$

3) Add mantissas

$Z = 1.0324 \times 10^3$

4) Normalize result

$Z = 0.1324 \times 10^4$



Implementing The Pipeline

- The comparator, shifter, adder/subtractor, incrementer and decrementer are implemented using combinational circuits.
- Assuming time delays of $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns and the registers have a delay $t_r = 10$ ns
- We choose a clock cycle of $t_p = t_3 + t_r = 110$
- A non-pipelined implementation would have a delay of $tn = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. This results in a speedup of $320/110 = 2.9$

Instruction Pipelining

- Pipeline processing can occur in the instruction stream as well, with the processor fetches instruction, while the previous instruction are being executed.

- It is possible for an instruction to cause a branch out of sequence, and the pipeline must be cleared of all the instructions after the branch.

- The instruction pipeline can read instructions from memory and place them in a queue when the processor is not accessing memory as part of the execution of instructions.

Steps in the Instruction Cycle

- Computers with complex instructions require several phases in order to process an instruction. These might include:

1. Fetch the instruction from memory
2. Decode the instruction
3. Calculate the effective address
4. Fetch the operands from memory
5. Execute the instruction
6. Store the result in the proper place

Difficulties Slowing Down the Instruction Pipeline

- Certain difficulties can prevent the instruction pipeline from operating at maximum speed:

  –Different segments may require different amounts of time (pipelining is most efficient with segments of the same duration)

  –Some segments are skipped for certain operations (e.g., memory reads for register-reference instructions)

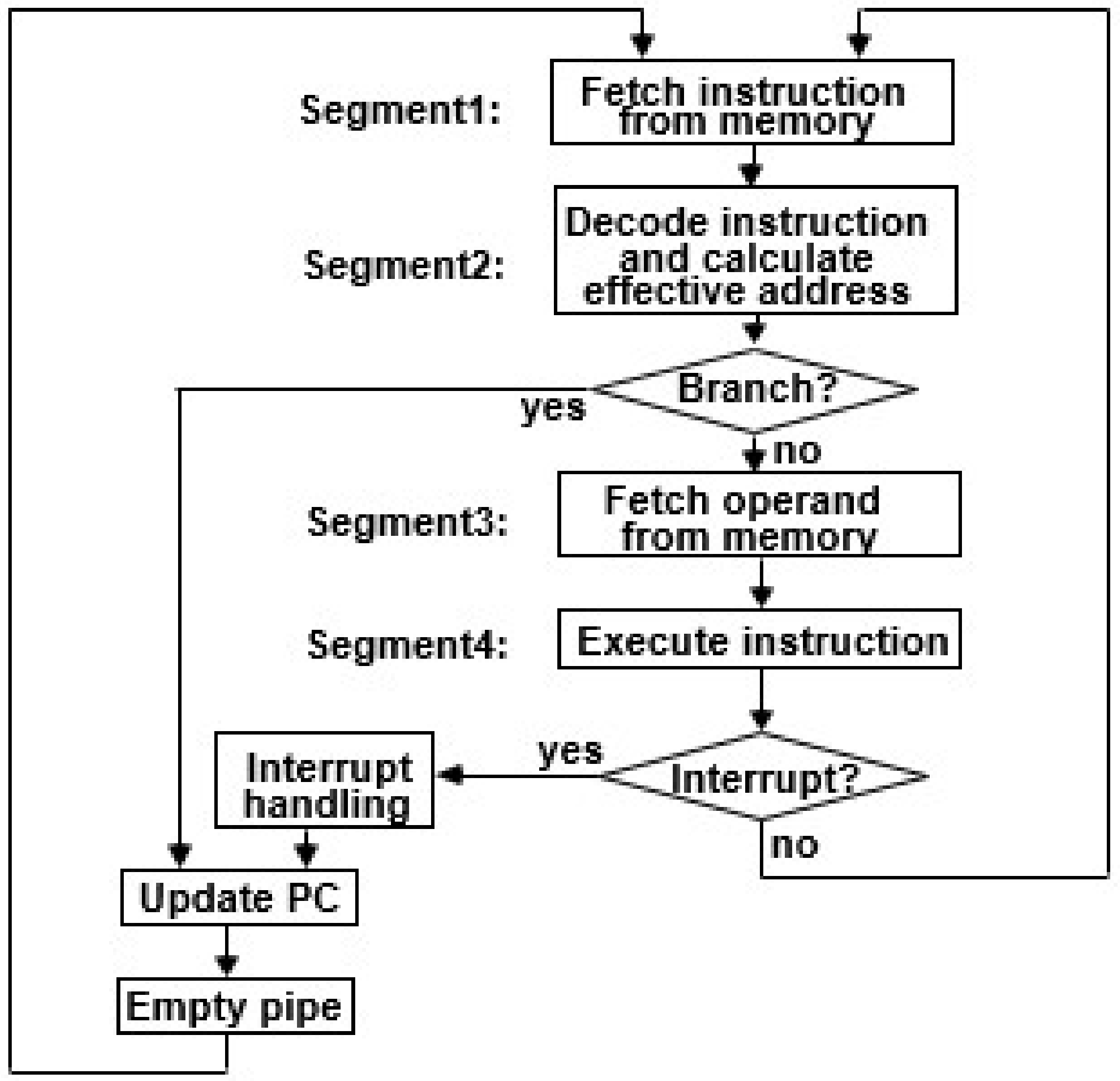
  –More that one segment may require memory access at the same time (this can be resolved with multiple busses).

Example: Four-Segment CPU Pipeline

- Assumptions:
  - Instruction decoding and effective address can be combined into one stage
  - Most of the instructions place the result into a register (execution and storing result become one stage)

    - Four-segment CPU pipeline :
      - 1) **FI** : Instruction Fetch
      - 2) **DA** : Decode Instruction & calculate EA
      - 3) **FO** : Operand Fetch
      - 4) **EX** : Execution

Four-segment CPU pipeline

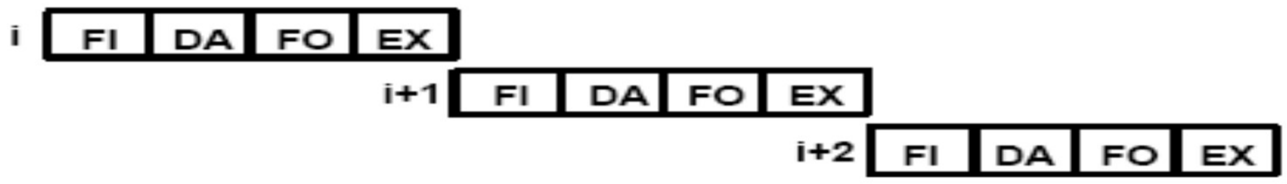


- While one instruction is being executed in segment 4, another is fetching an operand in segment 3, another is calculating an effective address and a fourth is being fetched.

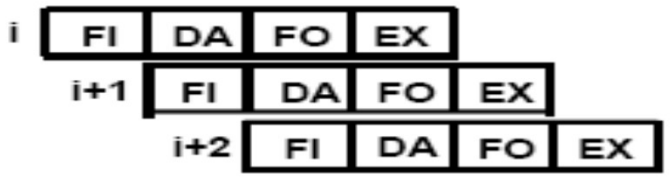
- If a branch is encountered, we complete the operations in the last segments, delete the instructions in the instruction buffer and restart the pipeline with the new address.

Execution of Three Instructions in a 4-Stage Pipeline

Conventional



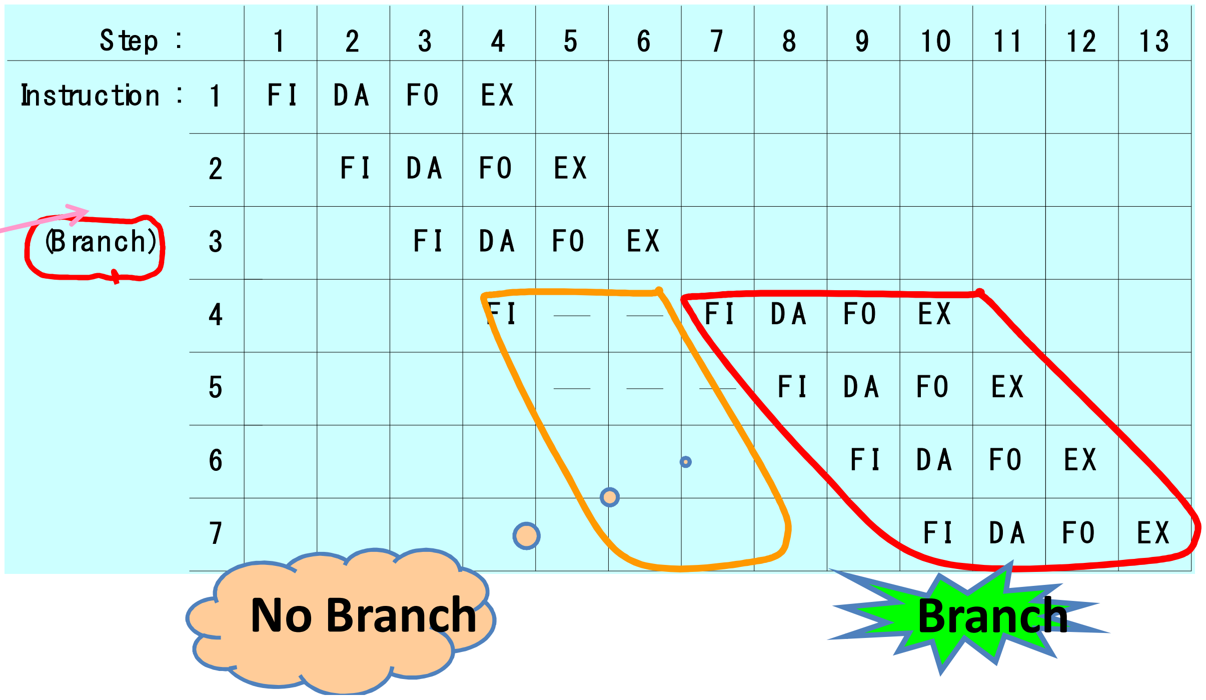
Pipelined



Timing of instruction pipeline

- Assume now that instruction 3 is a branch instruction.
- As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6.
- If the branch is taken, a new instruction is fetched in step 7.
- If the branch is not taken, the instruction fetched previously in step 4 can be used.
- The pipeline then continues until a new branch instruction is encountered.

» **Instruction 3 Branch**



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

Pipeline Conflicts

- There are three major difficulties that cause the instruction pipeline to deviate from its normal operations

1. **Resource conflicts** – caused by access to memory by two segments at the same time. Separate memory for data and instructions resolves this.
2. **Data dependency** – an instruction depends on the result of a previous instruction but this result is not yet available.
3. **Branch difficulties** – branch and other instructions(interrupt, ret) that change the PC's value.

Pipeline Hazards

Hazards: situations that prevent the next instruction from executing in the designated clock cycle.

3 classes of hazards:

structural hazard – resource conflicts
data hazard – data dependency
control hazard –Branch difficulties

Data Dependency

- A data dependency occurs when an instruction needs data that is not yet available.

- An instruction may need to fetch an operand being generated at the same time by an instruction that is being executed.

- An address dependency occurs when an address cannot be calculated because the necessary information is not yet available, e.g., an instruction with an register indirect address cannot fetch the operand because the address is not yet loaded into the register.

Dealing With Data Dependency

- Pipelined computers deal with data dependencies in several ways:
  - **Hardware interlocks** - a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. It delays the later instructions.
  - **Operand forwarding** – if a result is needed as a source in an instruction that is further down the pipeline, it is forwarded there, bypassing the register file. This requires special circuitry.
  - **Delayed load** – using a compiler that detects data dependencies in programs and adds NOP instructions to delay the loading of the conflicted data.

Handling Branch Instructions

- Branch instructions are a major problem in operating an instruction pipeline, whether they are ***unconditional*** (always occur) or ***conditional*** (depending on whether the condition is satisfied).

- These can be handled by several approaches:
  - **Prefetching target instruction** – Both the target instruction (of the branch) and the next instruction are fetched and saved until the branch is executed. This can be extended to include instructions after both places in memory.
  - **Use of a branch target buffer** – the target address and the instruction at that address are both stored in associative memory (along with the next few instructions).

Handling Branch Instructions (continued)

- **Using a loop buffer** – a small high-speed register file that stores the instructions of a program loop when it is detected.
- **Branch prediction** – guesses the outcome of a condition and prefetches instructions
- **Delayed branch** – used by most RISC processors. Branch instructions are detected and object code is rearranged by inserting instructions that keep the pipeline going. (This can be NOPs).

RISC Pipeline

- The RISC architecture is able to use an efficient pipeline that uses a small number of sub operations for several reasons:
  - Its fixed length format means that decoding can occur during register selection.
  - Data manipulation instructions executed using register-to-register operations, so there is no need to calculate effective addresses.

RISC Architecture and Memory-Reference Instructions

- The only data transfer instructions in RISC architecture are load and store, which use register indirect addressing, which typically requires 3-4 stages in the pipeline.
- Conflicts between instruction fetches and transferring the operand can be avoided with multiple busses that access separate memories for data and instructions.

Advantages of RISC Architecture

- RISC processors are able to execute instructions at a rate of one instruction per clock cycle.
  - Each instruction can be started with a new clock cycle and the processor is pipelined so that one clock cycle per instruction rate can be achieved.

- RISC processors can rely on the compiler to detect and minimize the delays caused by data conflicts and branch penalties.

Implementing the Pipeline

- The necessary hardware:
  - –The control unit fetches the instruction, saving it in an instruction register.
  - –The instruction is decoded while the registers are selected.
  - –The processor consists of registers and an ALU that does arithmetic, shifting and logic.
  - –Data memory is used for storing and loading data.

Example: 3-Segment Instruction Pipeline

- A typical RISC processor will have 3 instruction formats:
  - –Data manipulation instructions
  - –Data transfer instructions
  - –Program control instructions

» 1) **I** : Instruction fetch

» 2) **A** : Instruction decoded and ALU operation

» 3) **E** : Transfer the output of ALU to a register, memory, or PC

Delayed Load

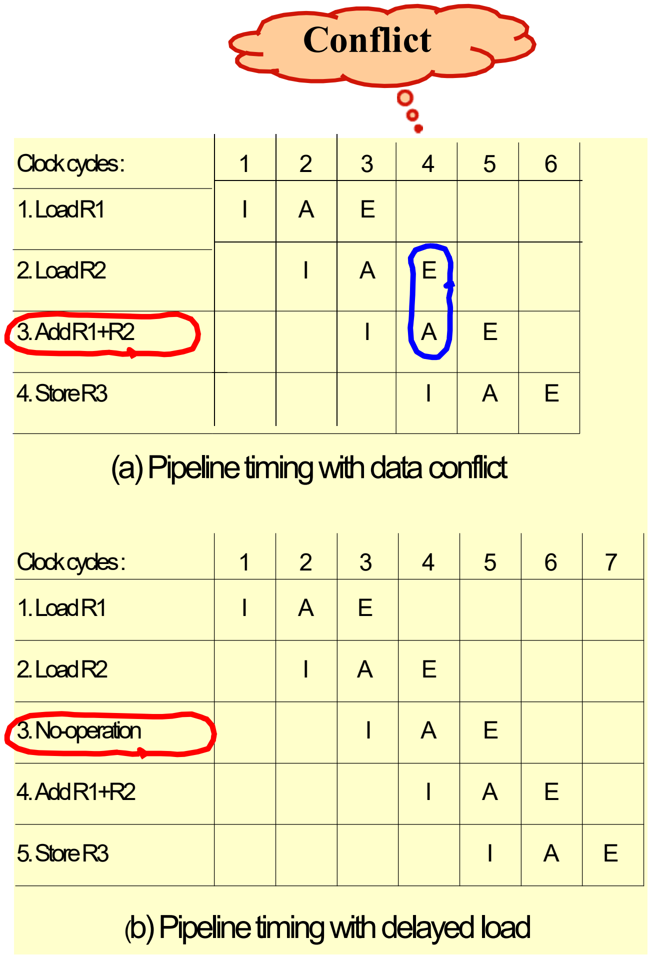
Consider now the operation of the following four instructions:

1. LOAD: Rl <--M [address 1]
2. LOAD: R2 <--M [address 2]
3. ADD: R3 <--Rl + R2
4. STORE: M[address 3] <--R3

» Instruction(**ADD R1 + R3**) Conflict

» Delayed Load

- No-operation



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

(a) Pipeline timing with data conflict

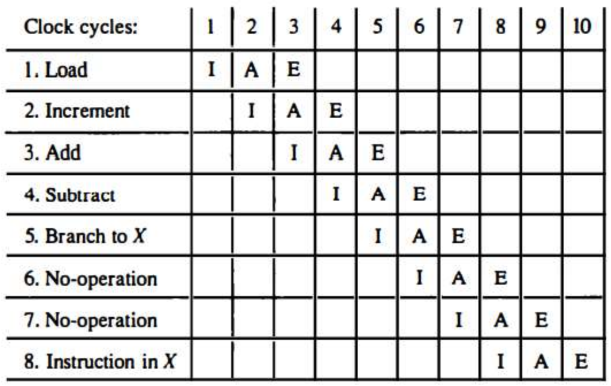
Clock cycles:	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

(b) Pipeline timing with delayed load

Delayed Branch

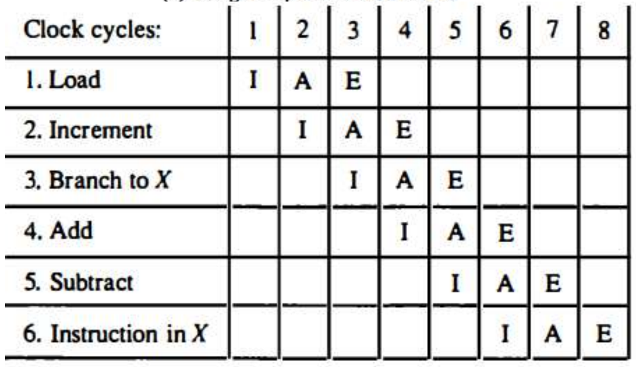
The example consists of five instructions:

1. Load from memory to R1
2. Increment R2
3. Add R3 to R4
4. Subtract R5 from R6
5. Branch to address X



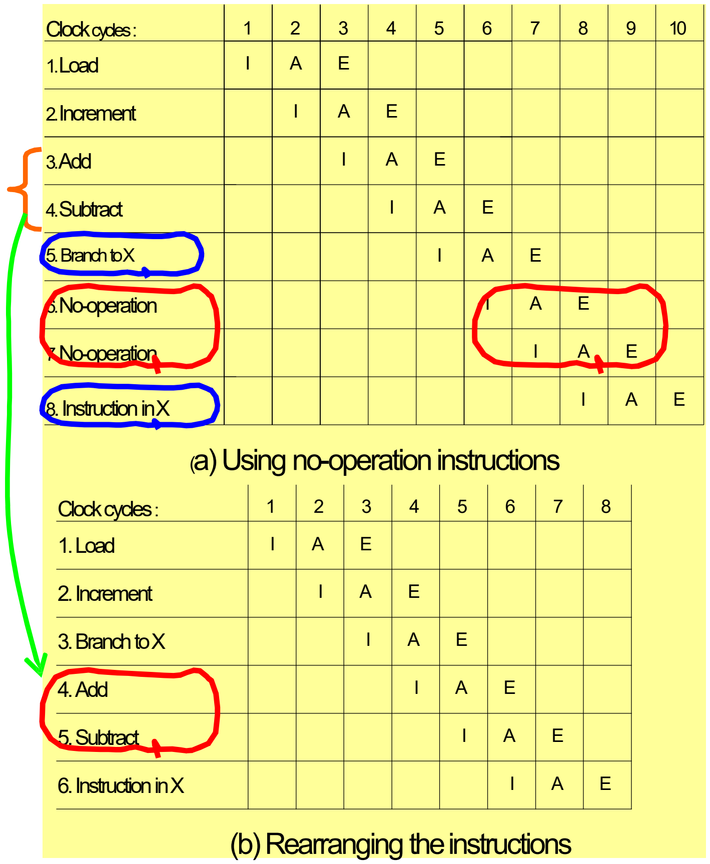
Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to *X*					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in *X*								I	A	E

(a) Using no-operation instructions



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

(b) Rearranging the instructions



Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

(a) Using no-operation instructions

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

(b) Rearranging the instructions

Vector Processing

- Science and Engineering Applications
  - Long-range weather forecasting, Petroleum explorations, Seismic data analysis, Medical diagnosis, Aerodynamics and space flight simulations, Artificial intelligence and expert systems, Mapping the human genome, Image processing
- Vector Operations
  - Arithmetic operations on large arrays of numbers
  - Conventional scalar processor
    - » Machine language

```
    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
```

    - » Fortran language

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

  - Vector processor
    - » Single vector instruction

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

- A vector is an ordered set of data items in a one-dimensional array.
  - A vector V of length n can be represented as a row vector by $V = [V_1, V_2, \dots, V_n]$
  - If these values were listed in a column, it would be a column vector.

- **Vector Instruction Format :**

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
ADD	*A*	*B*	*C*	*100*

- **Matrix Multiplication**
  - 3 x 3 matrices multiplication : **n²** = **9** inner products

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

    » $c_{11} = a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31}$

  - Cumulative multiply-add operation : **n³** = **27** multiply-add

$c = c + a \times b$

    » $c_{11} = c_{11} + a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31}$: multiply-add

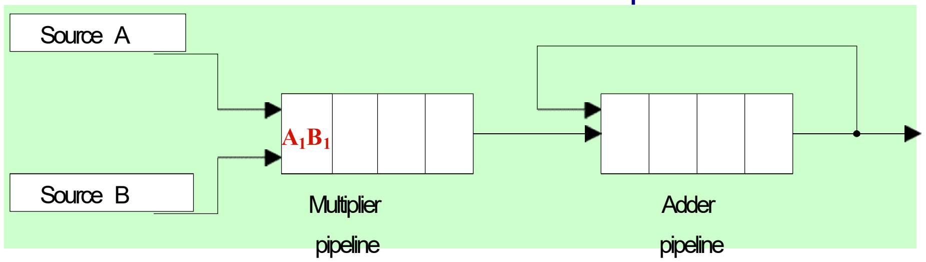
9 X 3 multiply-add = **27**

Initialize $C_{11} = 0$

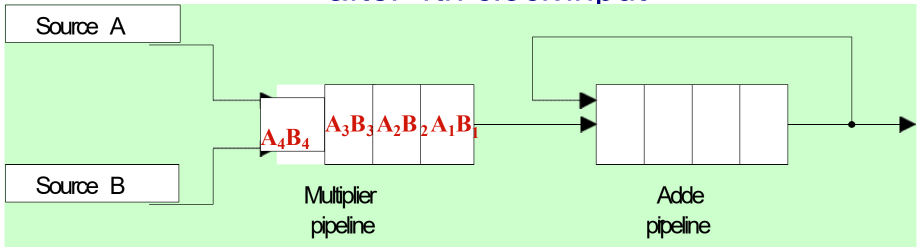
Pipeline for calculating an inner product :

- Floating point multiplier pipeline : 4 segment
- Floating point adder pipeline : 4 segment
- $C = A_1B_1 + A_2B_2 + A_3B_3 + . . \quad + A_kB_k$

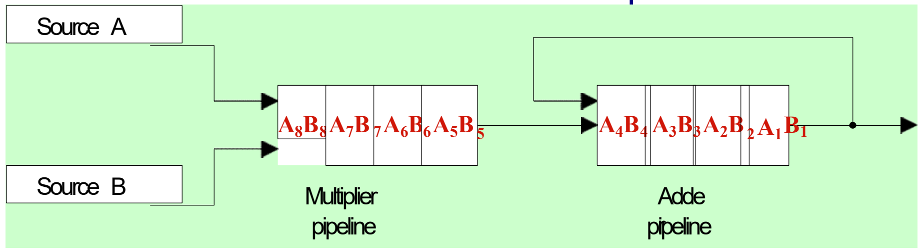
» after 1st clock input



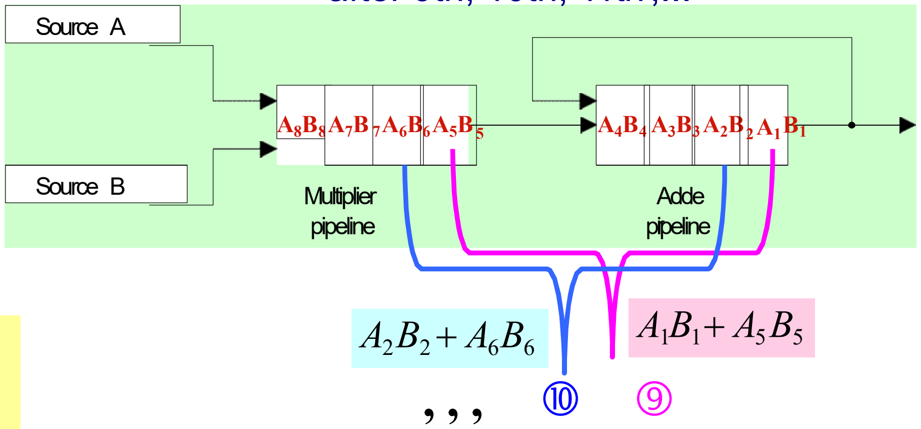
» after 4th clock input



» after 8th clock input



» after 9th, 10th, 11th ,...



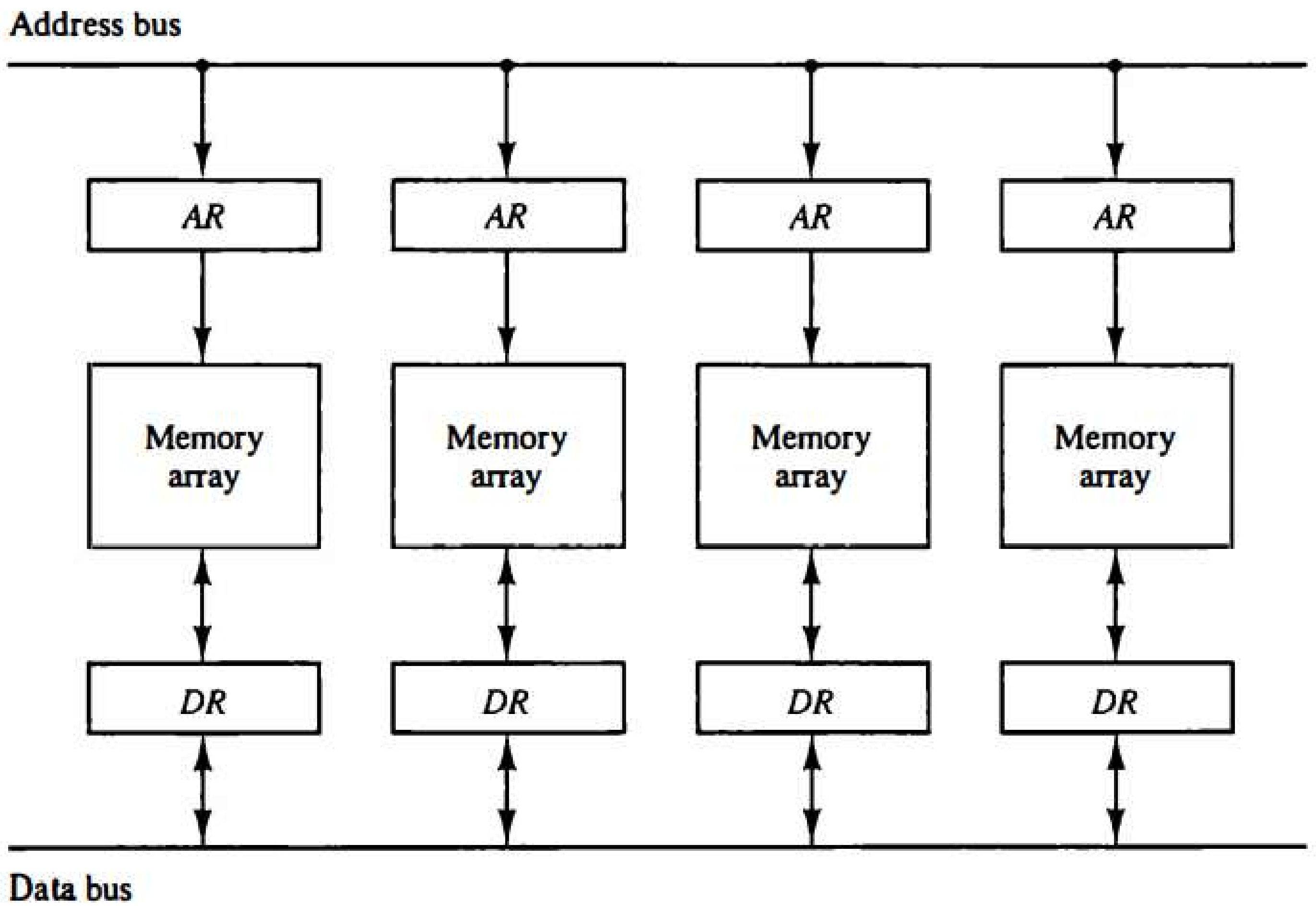
» Four section summation

$$
\begin{aligned}
C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \dots \\
& + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \dots \\
& + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \dots \\
& + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \dots
\end{aligned}
$$

Why Multiple Module Memory?

- Pipeline and vector processors frequently need access to memory from multiple sources at the same time.
  - Instruction pipelines may need to fetch an instruction and an operand at simultaneously.
  - An arithmetic pipeline may need more than operand at the same time.
- Instead of using multiple memory busses, memory can be partitioned into separate modules.

Multiple Module Memory Organization



Memory Interleaving

- Multiple memory units allow the use of ***memory interleaving***, where different sets of addresses are assigned to different modules.

- *n*-way interleaved memory fetches can be staggered, reducing the effective memory cycle time by factor that is close to *n*.

Supercomputers

- Supercomputer = Vector Instruction + Pipelined floating-point arithmetic
- Performance Evaluation Index
  - » **MIPS** : Million Instruction Per Second
  - » **FLOPS** : Floating-point Operation Per Second
    - megaflops : 10^6, gigaflops : 10^9
- Cray supercomputer : Cray Research
  - » Cray-1 : 80 megaflops, 4 million 64 bit words memory
  - » Cray-2 : 12 times more powerful than the cray-1
- VP supercomputer : Fujitsu
  - » VP-200 : 300 megaflops, 32 million memory, 83 vector instructions, 195 scalar instructions
  - » VP-2600 : 5 gigaflops

Array Processors

- Array processors performs computations on large arrays of data.

- There are two different types of such processors:
  - Attached array processors, which are auxiliary processors attached to a general-purpose computer.
  - SIMD array processors, which are processors with an SIMD organization that uses multiple functional units to perform vector operations.

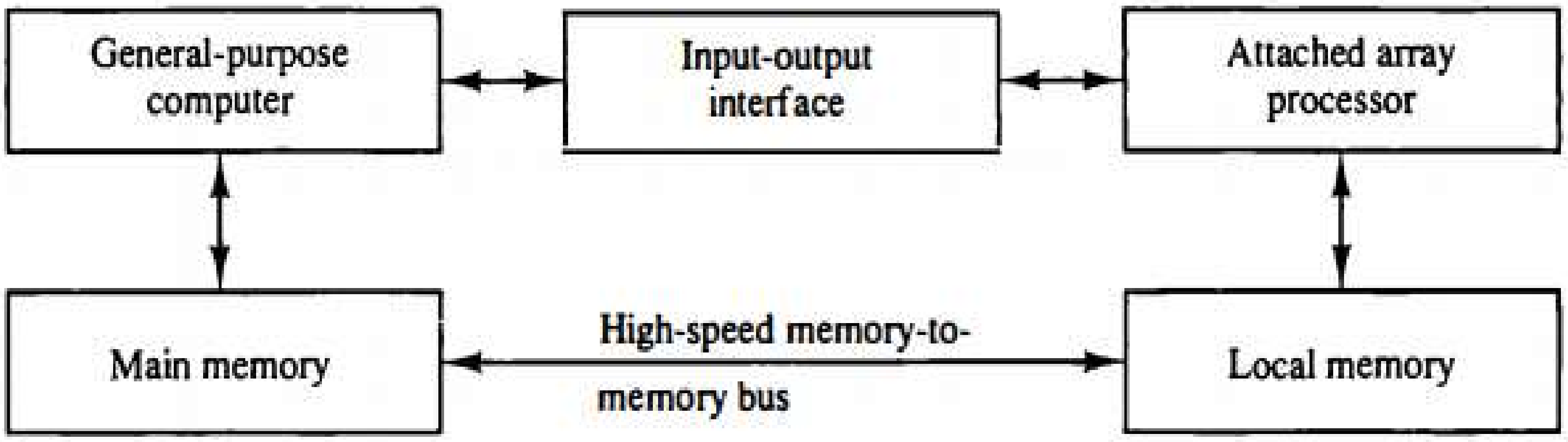
Attached Array Processor

- An attached array processor is designed as a peripheral for a conventional host computer, providing vector processing for complex scientific applications.

- The array processor, working with multiple functional units, serves as a back-end machine driven by the 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 supercomputers.

Figure 9-14 Attached array processor with host computer.



SIMD Array Processor

- An SIMD processor is computer with multiple processing units running in parallel.

- The processing units are synchronized to perform the same operation under a common control unit on multiple data streams.
  - Each processing element has its own ALU, FPU and working registers as well as local memory.

- The master control unit main purpose is to decode instruction and determine how they are executed.

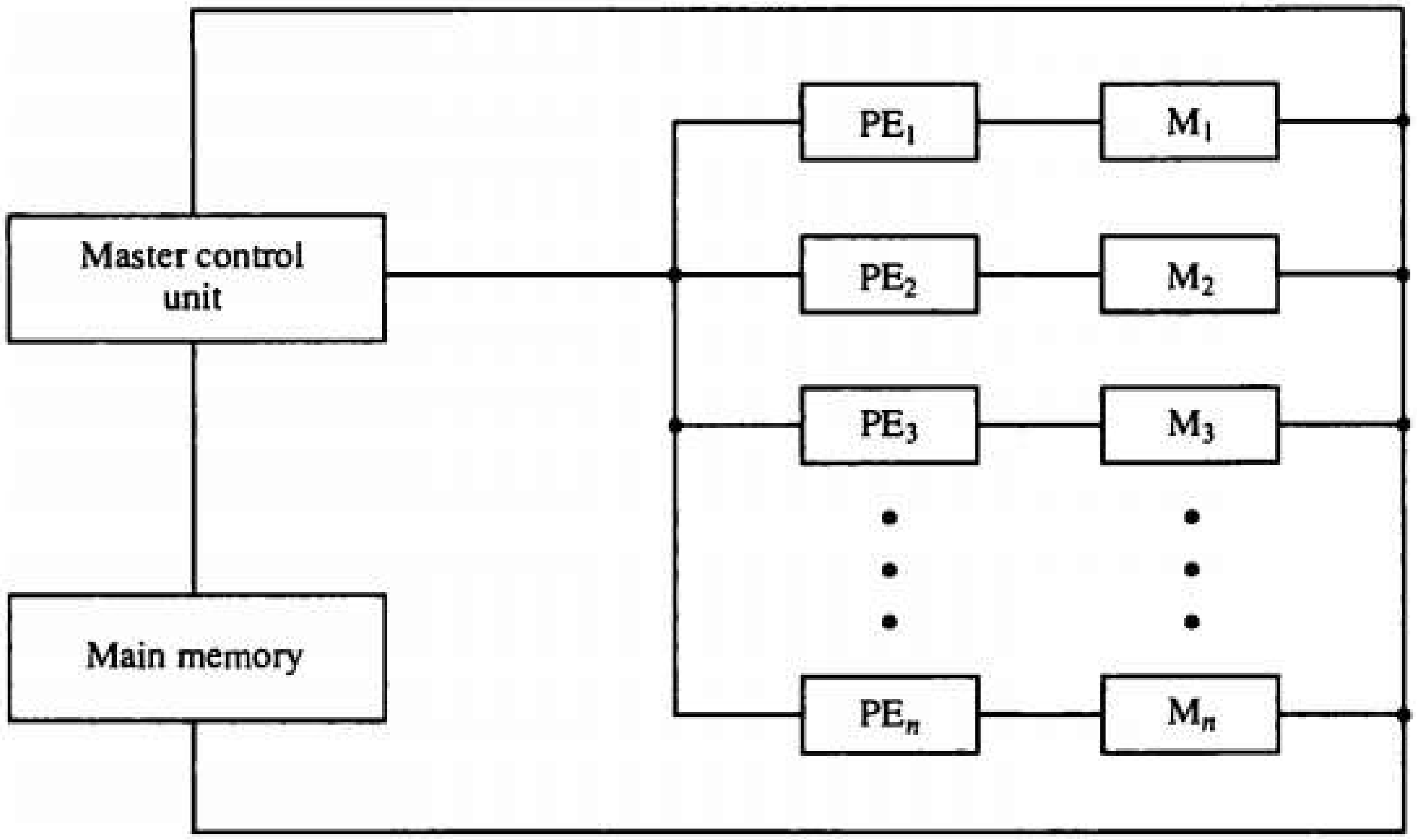
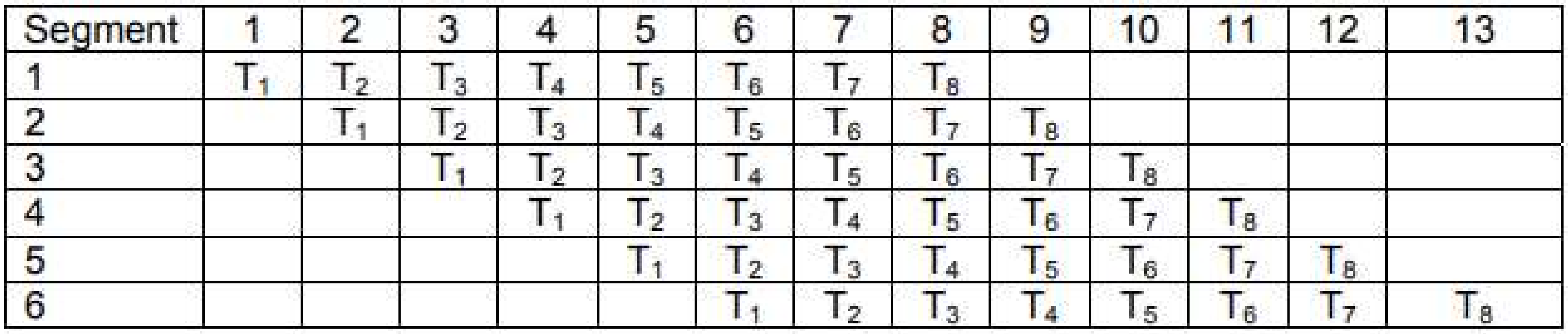


Figure 9-15 SIMD array processor organization.

Example

- The vector addition $C = A + B$.
- The master control unit first stores the ith components a_i and b_i of A and B in local memory M_i for $i = 1, 2, 3, . . . , n$.
- It then broadcasts the floating-point add instruction $c_i = a_i + b_i$ to all PEs, causing the addition to take place simultaneously.
- The components of c_i are stored in fixed locations in each local memory.
- This produces the desired vector sum in one add cycle.

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



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

2. Determine the number of dock cycles that it takes to process 200 tasks in a six-segment pipeline

k = 6 segments

n = 200 tasks (k + n – 1)$\underset{p}{k}$ = 6 + 200 – 1 = 205 cycles

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

$t_n = 50$ ns
$k = 6$
$t_p = 10$ ns
$n = 100$

$$S = \frac{nt_n}{(k+n-1)t_p} = \frac{100 \times 50}{(6-99) \times 10} = 4.76$$

$$S_{\max} = \frac{t_n}{t_p} = \frac{50}{10} = 5$$

4. Consider the multiplication of two 40 x 40 matrices using a vector processor.

a. How many product terms are there in each inner product, and how many inner products must be evaluated?

There are 40 product terms in each inner product,

40^2 = 1,600 inner products

b. How many multiply-add operations are needed to calculate the product matrix?

40^3 – 64,000