

# **PIPELINING AND VECTOR PROCESSING**

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

# PARALLEL PROCESSING

**Execution of *Concurrent Events* in the computing process to achieve faster *Computational Speed***

## **Levels of Parallel Processing**

- Job or Program level**
  
- Task or Procedure level**
  
- Inter-Instruction level**
  
- Intra-Instruction level**

# PARALLEL COMPUTERS

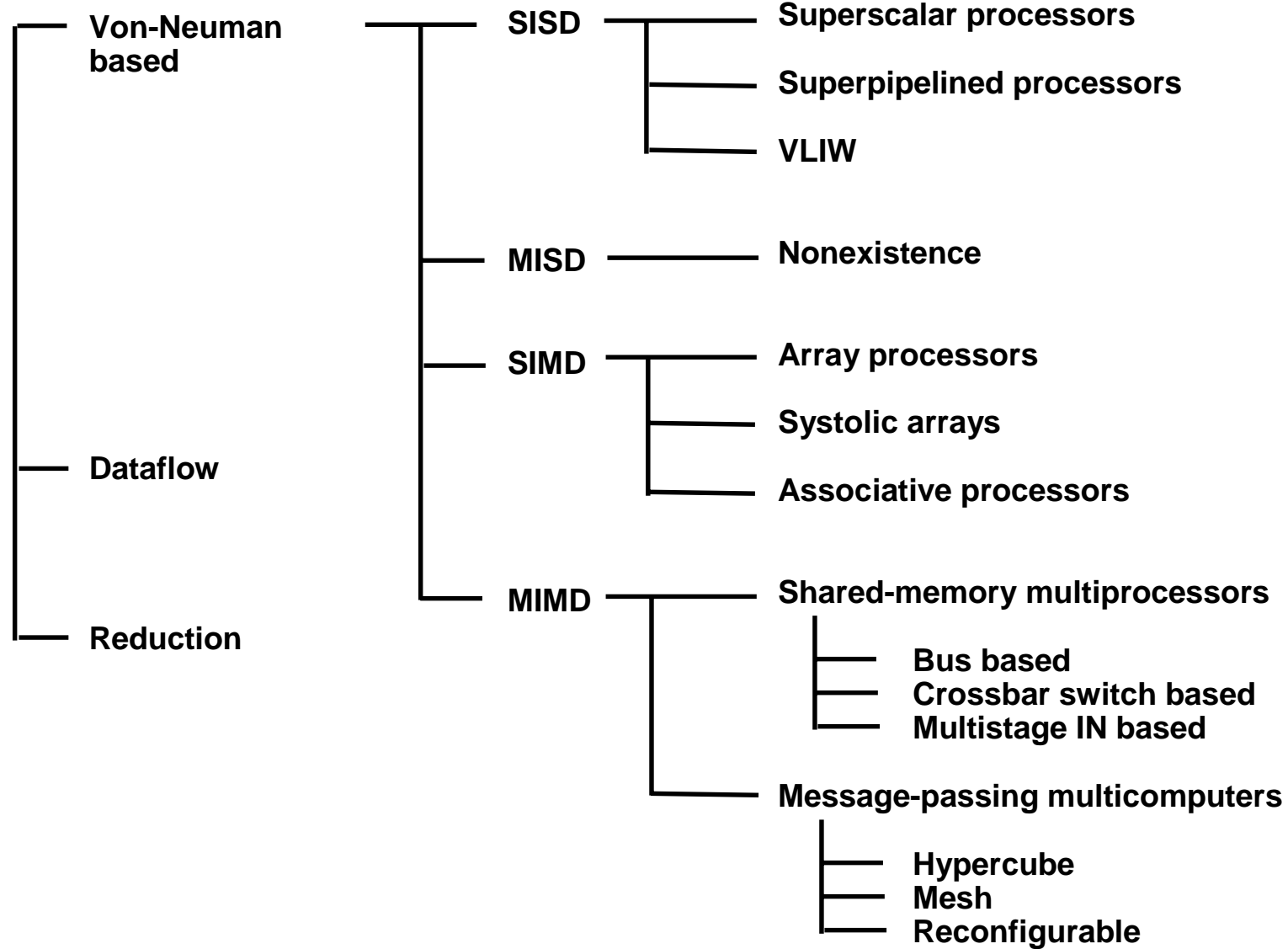
## Architectural Classification

\* Flynn's classification

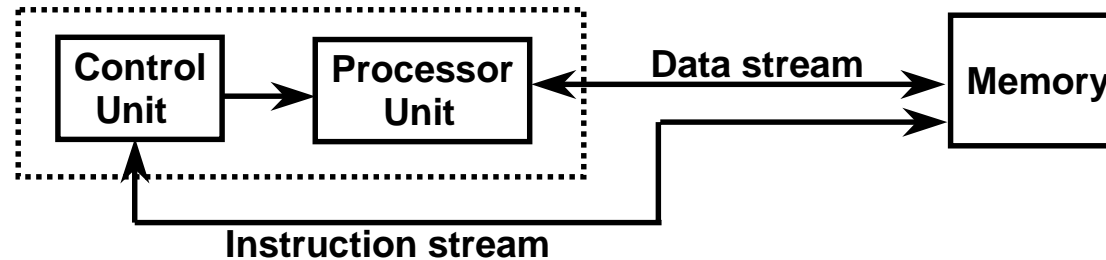
- Based on the multiplicity of *Instruction Streams* and *Data Streams*
- Instruction Stream  
     Sequence of Instructions read from memory
- Data Stream  
     Operations performed on the data in the processor

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

# COMPUTER ARCHITECTURES FOR PARALLEL PROCESSING



# SISD COMPUTER SYSTEMS



## Characteristics

- Standard von Neumann machine
- Instructions and data are stored in memory
- One operation at a time

## Limitations

Von Neumann bottleneck

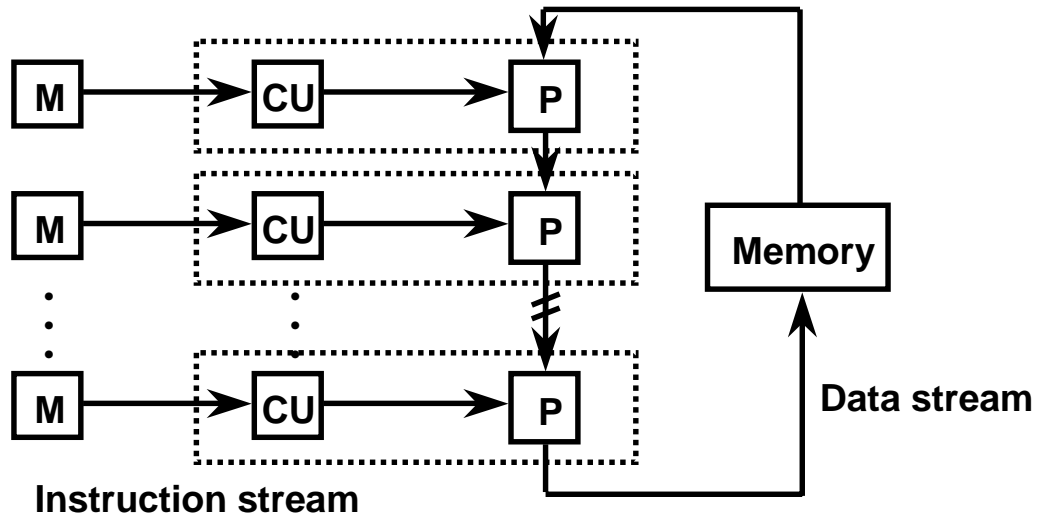
Maximum speed of the system is limited by the *Memory Bandwidth* (bits/sec or bytes/sec)

- Limitation on *Memory Bandwidth*
- Memory is shared by CPU and I/O

## **PERFORMANCE IMPROVEMENTS**

- **Multiprogramming**
- **Spooling**
- **Multifunction processor**
- **Pipelining**
- **Exploiting instruction-level parallelism**
  - **Superscalar**
  - **Superpipelining**
  - **VLIW (Very Long Instruction Word)**

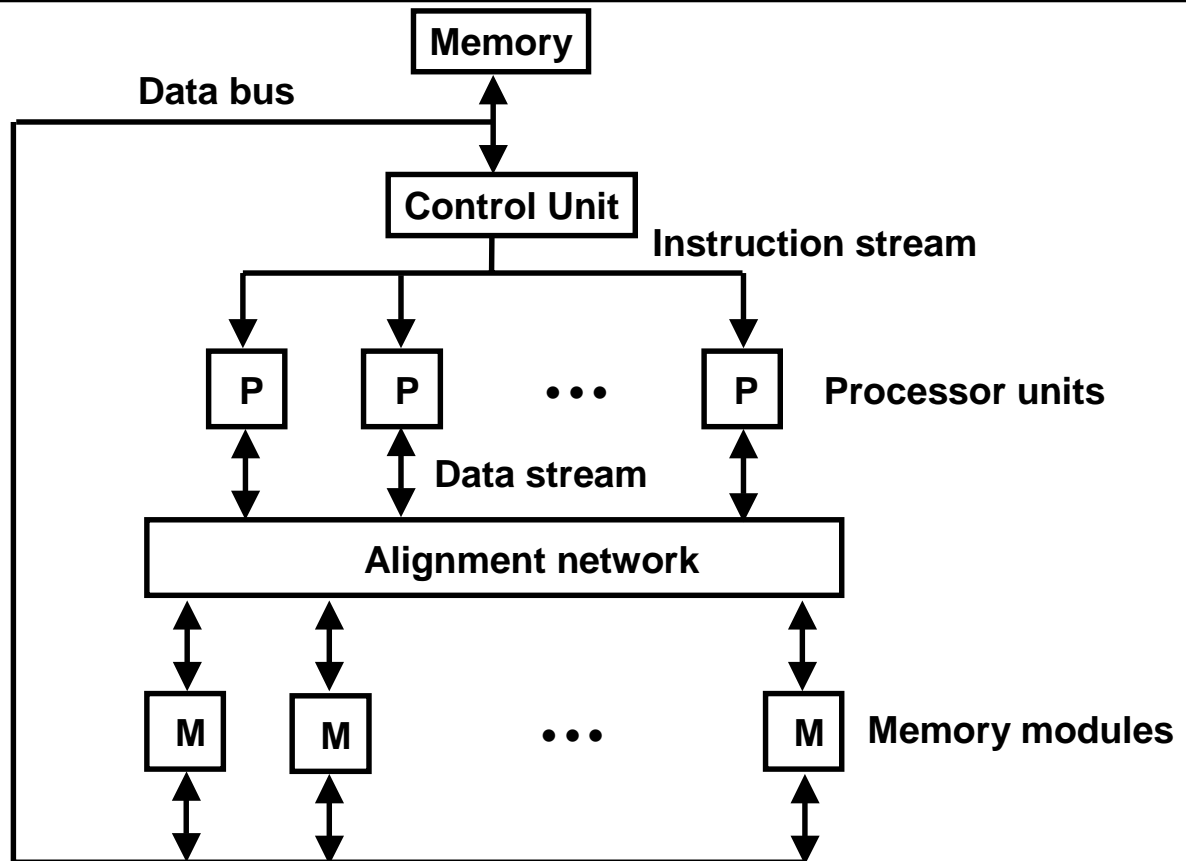
# MISD COMPUTER SYSTEMS



## Characteristics

- There is no computer at present that can be classified as MISD

# SIMD COMPUTER SYSTEMS



## Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time



## **TYPES OF SIMD COMPUTERS**

### **Array Processors**

- The control unit broadcasts instructions to all PEs, and all active PEs execute the same instructions
- ILLIAC IV, GF-11, Connection Machine, DAP, MPP

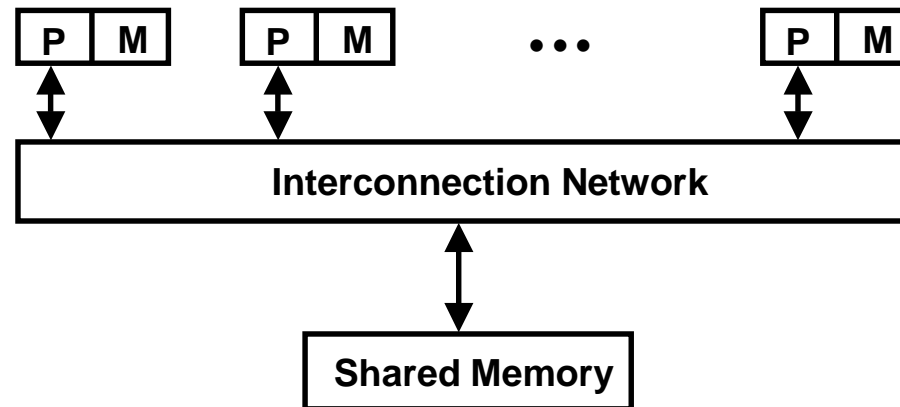
### **Systolic Arrays**

- Regular arrangement of a large number of very simple processors constructed on VLSI circuits
- CMU Warp, Purdue CHiP

### **Associative Processors**

- Content addressing
- Data transformation operations over many sets of arguments with a single instruction
- STARAN, PEPE

# MIMD COMPUTER SYSTEMS



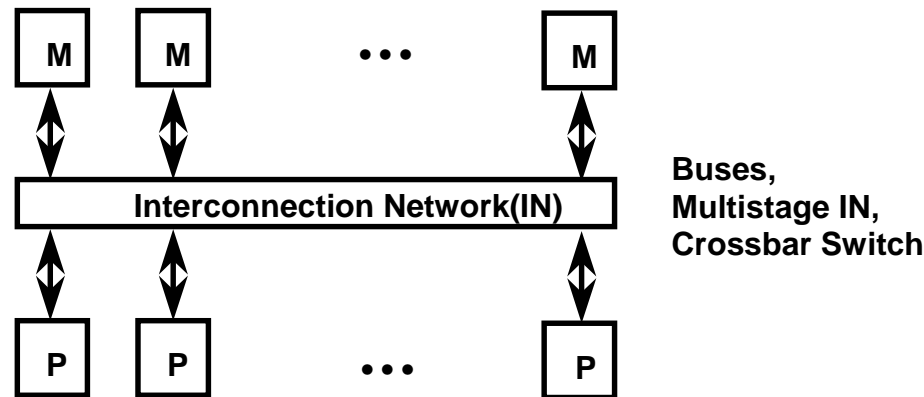
## Characteristics

- Multiple processing units
- Execution of multiple instructions on multiple data

## Types of MIMD computer systems

- Shared memory multiprocessors
- Message-passing multicomputers

# SHARED MEMORY MULTIPROCESSORS



## Characteristics

All processors have equally direct access to one large memory address space

## Example systems

Bus and cache-based systems

- Sequent Balance, Encore Multimax

Multistage IN-based systems

- Ultracomputer, Butterfly, RP3, HEP

Crossbar switch-based systems

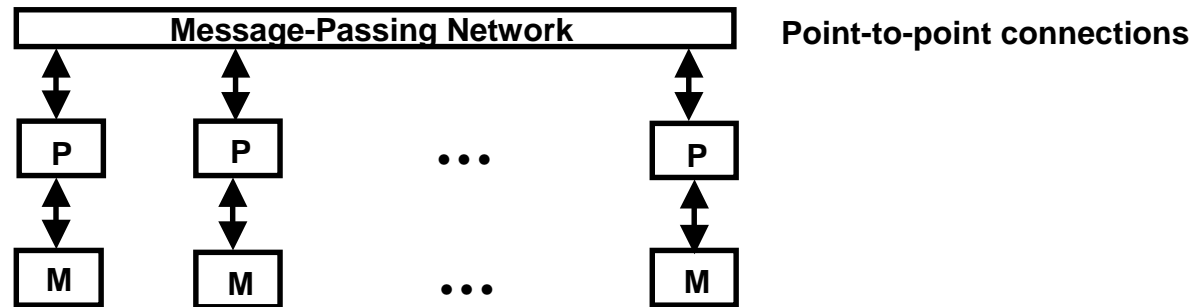
- C.mmp, Alliant FX/8

## Limitations

Memory access latency

Hot spot problem

## MESSAGE-PASSING MULTICOMPUTER



### Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

### Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

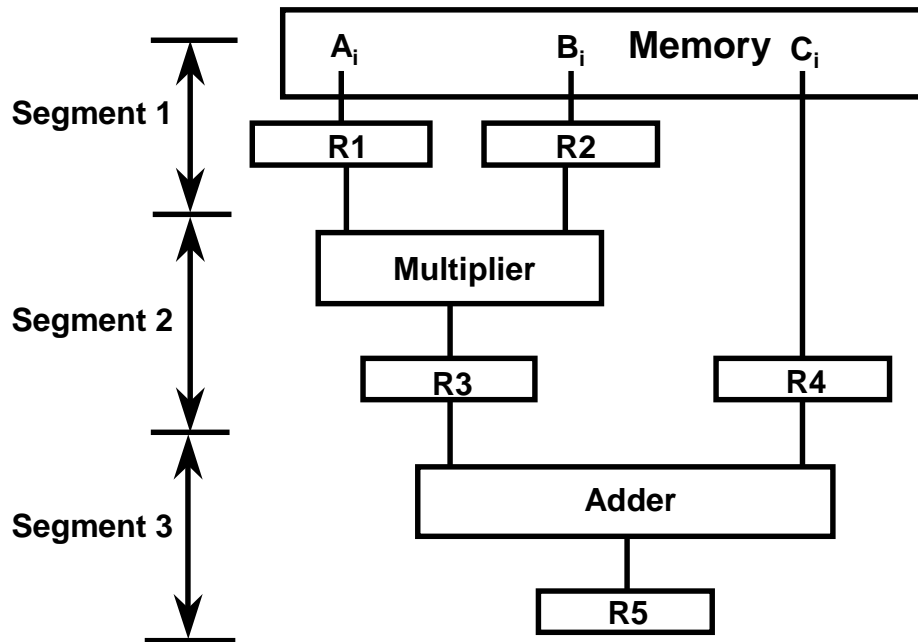
### Limitations

- Communication overhead
- Hard to programming

# PIPELINING

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

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \dots, 7$$



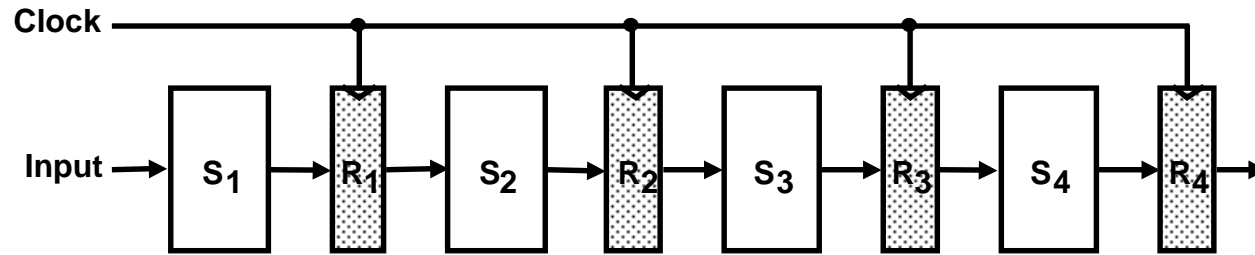
$R1 \leftarrow A_i$	$R2 \leftarrow B_i$	Load $A_i$ and $B_i$
$R3 \leftarrow R1 * R2$	$R4 \leftarrow C_i$	Multiply and load $C_i$
$R5 \leftarrow R3 + R4$		Add

## OPERATIONS IN EACH PIPELINE STAGE

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

# GENERAL PIPELINE

## General Structure of A 4-Segment Pipeline



## Space-Time Diagram

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

## PIPELINE SPEEDUP

**n:** Number of tasks to be performed

### Conventional Machine (Non-Pipelined)

**tn:** Clock cycle

**$\tau_1$ :** Time required to complete the n tasks

$$\tau_1 = n * t_n$$

### Pipelined Machine (k stages)

**tp:** Clock cycle (time to complete each suboperation)

**$\tau_k$ :** Time required to complete the n tasks

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

### Speedup

**Sk:** Speedup

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

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



# PIPELINE AND MULTIPLE FUNCTION UNITS

## Example

- 4-stage pipeline
- suboperation in each stage;  $t_p = 20\text{nS}$
- 100 tasks to be executed
- 1 task in non-pipelined system;  $20 * 4 = 80\text{nS}$

## Pipelined System

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

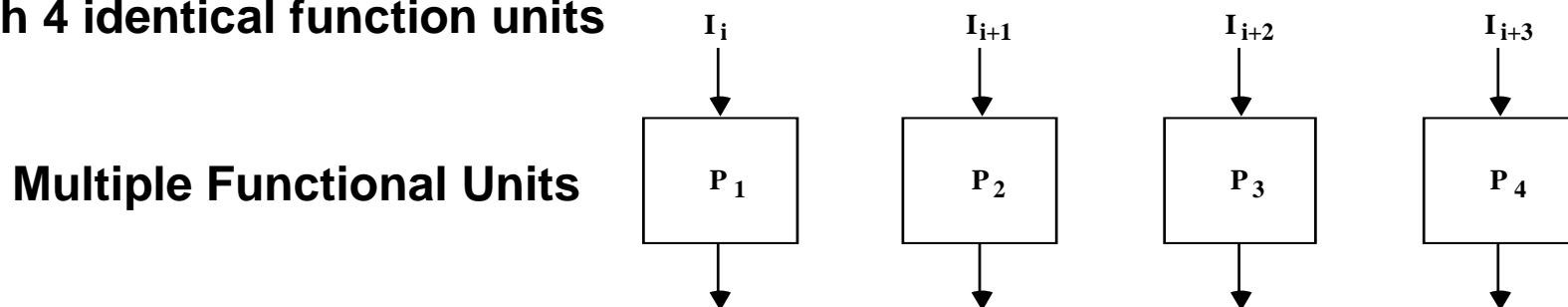
## Non-Pipelined System

$$n * k * t_p = 100 * 80 = 8000\text{nS}$$

## Speedup

$$S_k = 8000 / 2060 = 3.88$$

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



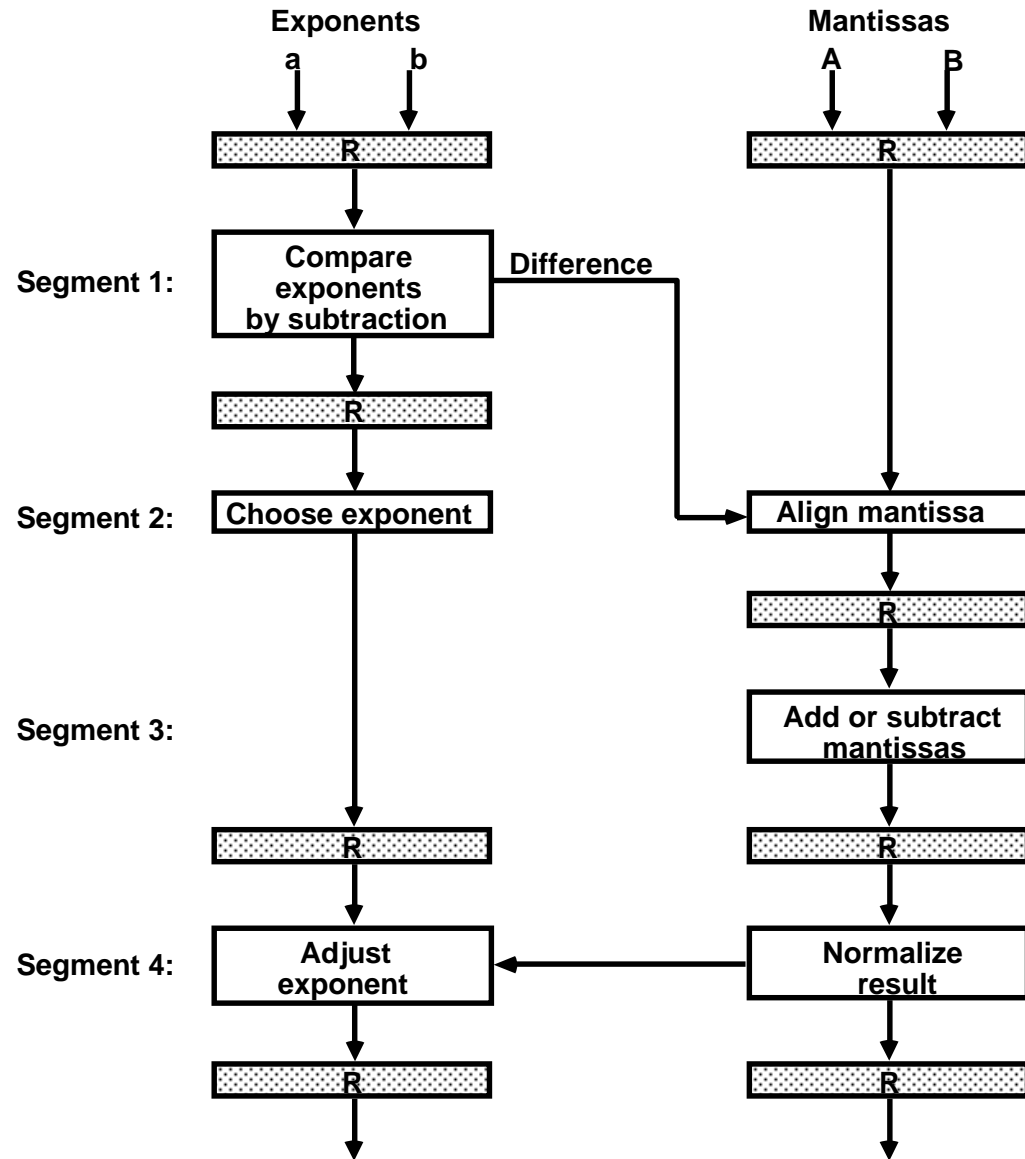
# ARITHMETIC PIPELINE

## Floating-point adder

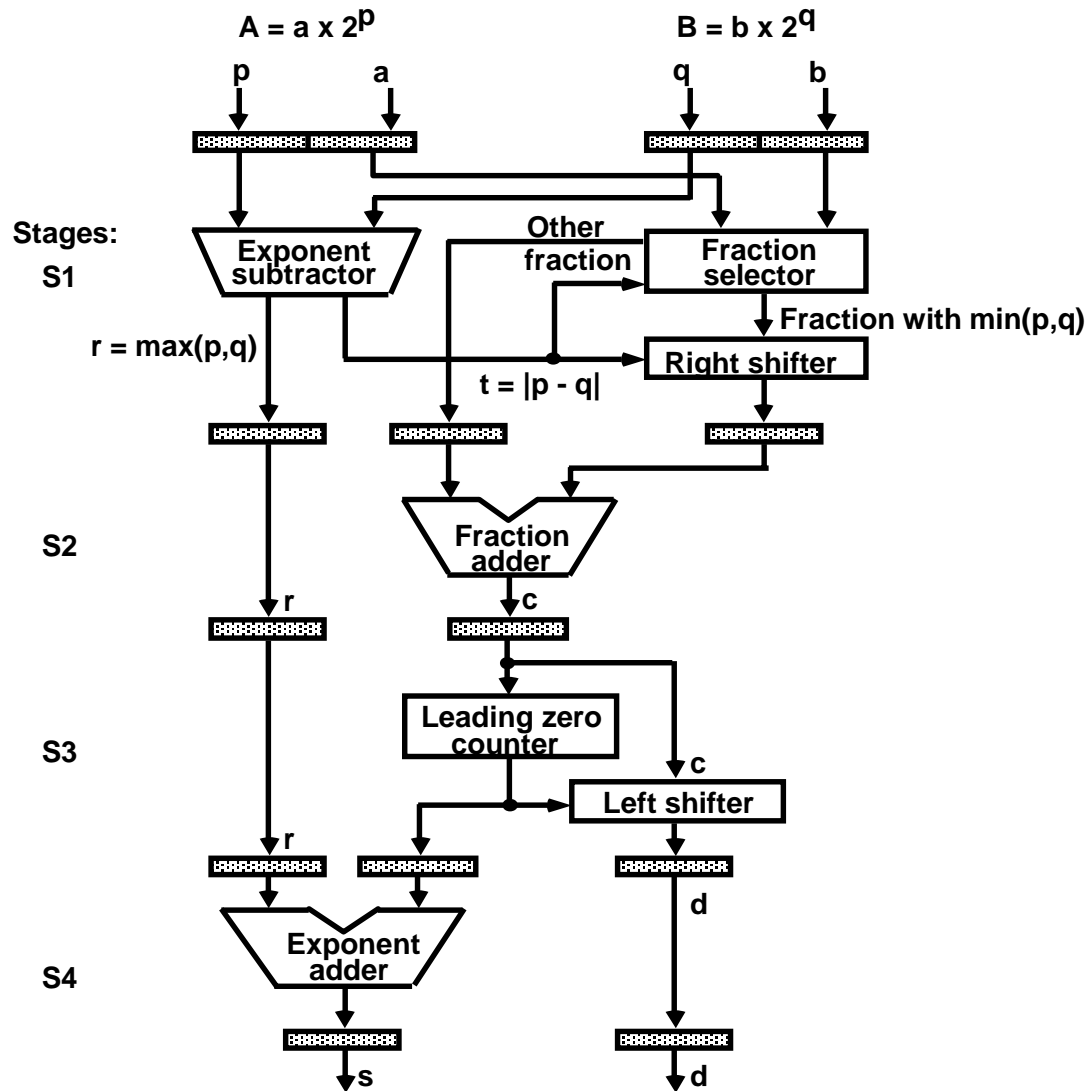
$$X = A \times 2^a$$

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

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



# 4-STAGE FLOATING POINT ADDER



$$C = A + B = c \times 2^r = d \times 2^s$$

$$(r = \max(p, q), 0.5 \leq d < 1)$$

## INSTRUCTION CYCLE

### Six Phases\* in a Instruction Cycle

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

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

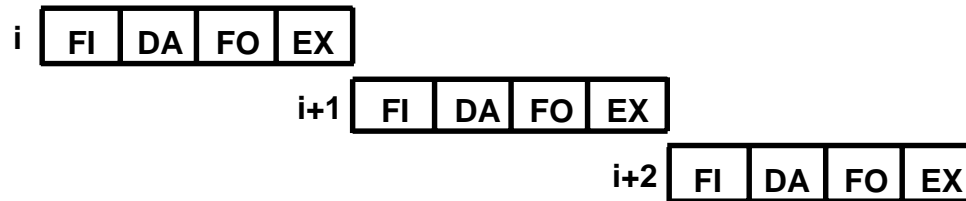
### ==> 4-Stage Pipeline

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

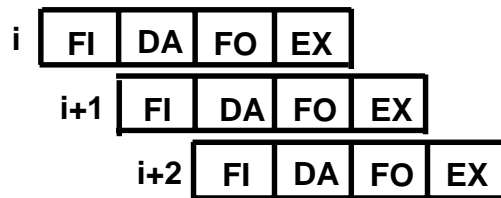
# INSTRUCTION PIPELINE

## Execution of Three Instructions in a 4-Stage Pipeline

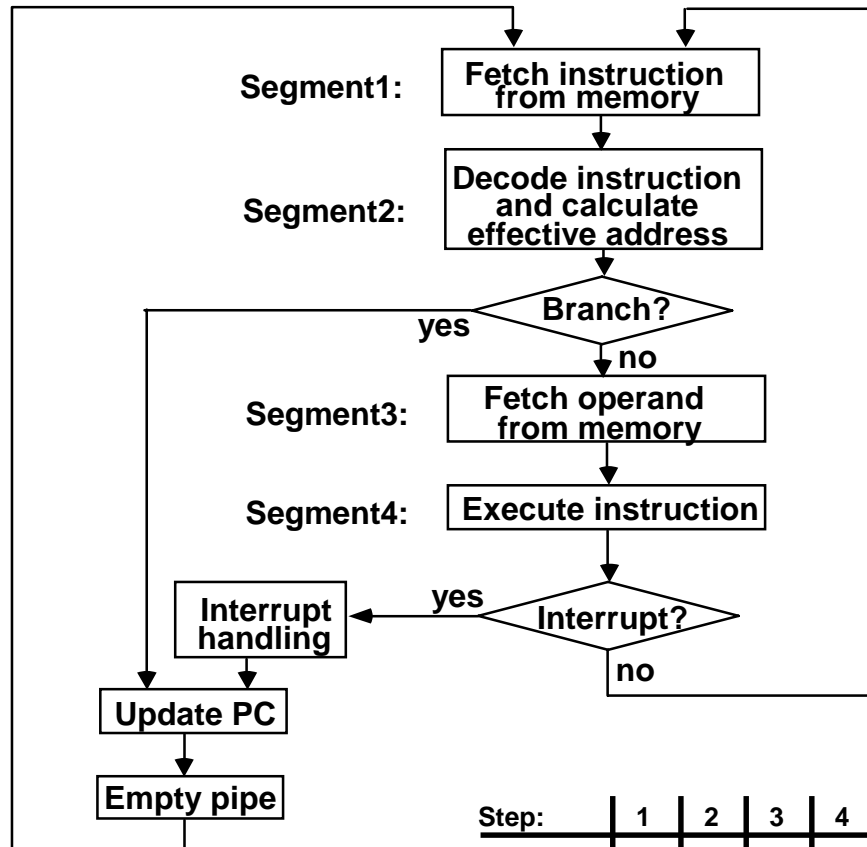
### Conventional



### Pipelined



# INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



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

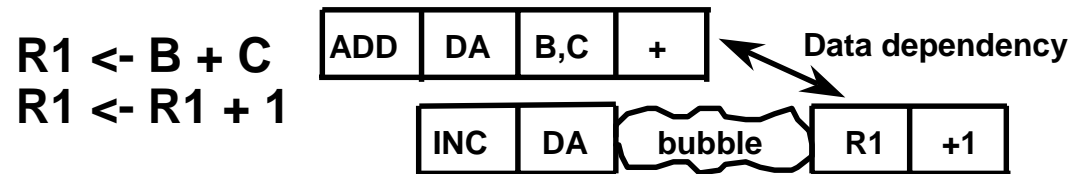
# MAJOR HAZARDS IN PIPELINED EXECUTION

## Structural hazards(Resource Conflicts)

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

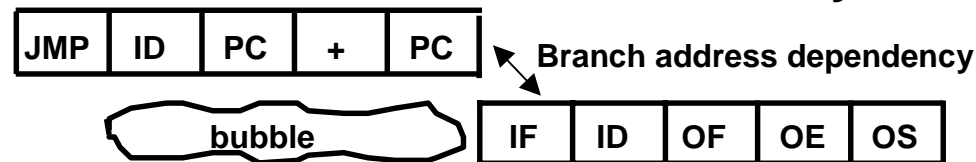
## Data hazards (Data Dependency Conflicts)

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



## Control hazards

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



Hazards in pipelines may make it necessary to **stall** the pipeline



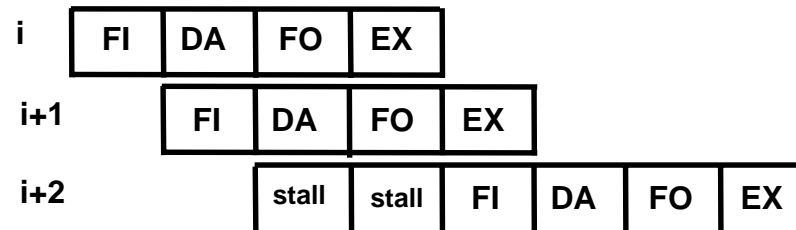
Pipeline Interlock:  
Detect Hazards Stall until it is cleared

# STRUCTURAL HAZARDS

## Structural Hazards

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

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



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



## DATA HAZARDS

### Data Hazards

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

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

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

### Hardware Technique

#### *Interlock*

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

#### *Forwarding* (bypassing, short-circuiting)

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

### Software Technique

Instruction Scheduling(compiler) for *delayed load*

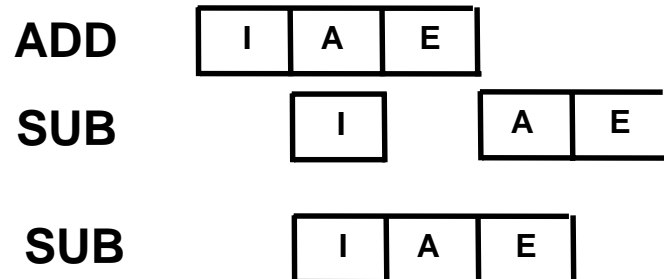
# FORWARDING HARDWARE

Example:

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

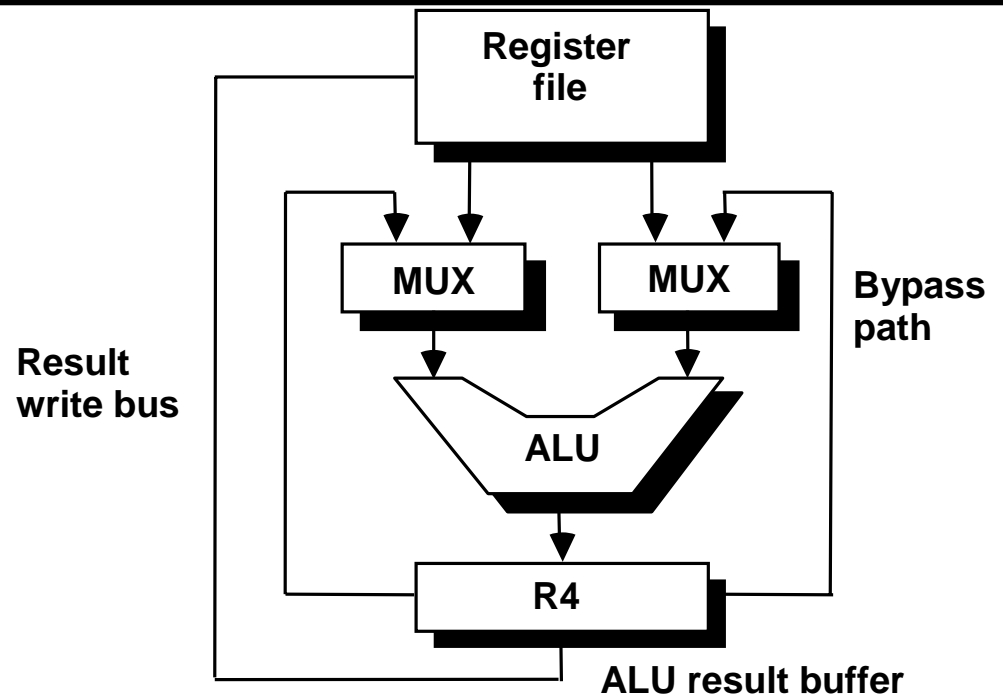
3-stage Pipeline

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



Without Bypassing

With Bypassing



# INSTRUCTION SCHEDULING

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

### Unscheduled code:

```

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

### Scheduled Code:

```

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

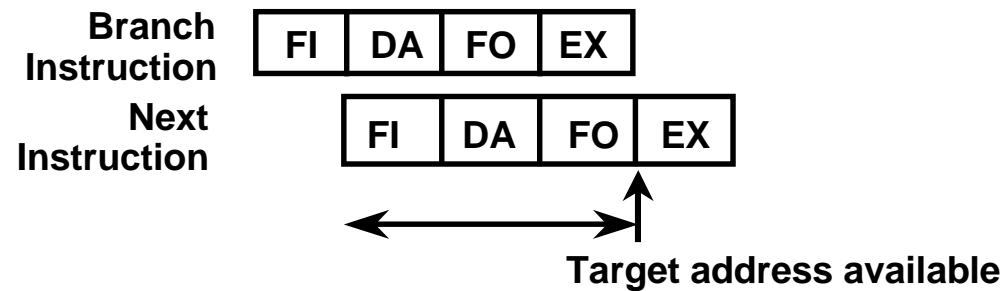
### Delayed Load

A load requiring that the following instruction not use its result

# CONTROL HAZARDS

## Branch Instructions

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



- Stall -> waste of cycle times

## Dealing with Control Hazards

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

## CONTROL HAZARDS

### Prefetch Target Instruction

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

### Branch Target Buffer(BTB; Associative Memory)

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

### Loop Buffer(High Speed Register file)

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

### Branch Prediction

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

### Delayed Branch

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

# RISC PIPELINE

## RISC

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

## Instruction Cycles of Three-Stage Instruction Pipeline

### Data Manipulation Instructions

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

### Load and Store Instructions

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

### Program Control Instructions

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

## DELAYED LOAD

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

### Three-segment pipeline timing

#### Pipeline timing with data conflict

clock cycle	1	2	3	4	5	6
Load R1	I	A	E			
Load R2		I	A	E		
Add R1+R2			I	A	<b>E</b>	
Store R3				I	<b>A</b>	E

#### Pipeline timing with delayed load

clock cycle	1	2	3	4	5	6	7
Load R1	I	A	E				
Load R2		I	A	E			
NOP			I	A	E		
Add R1+R2				I	A	E	
Store R3					I	A	E

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

# DELAYED BRANCH

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

## Using no-operation instructions

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

## Rearranging the instructions

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



# VECTOR PROCESSING

## Vector Processing Applications

- **Problems that can be efficiently formulated in terms of vectors**
  - 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 Processor (computer)

**Ability to process vectors, and related data structures such as matrices and multi-dimensional arrays, much faster than conventional computers**

**Vector Processors may also be pipelined**

## VECTOR PROGRAMMING

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

### Conventional computer

```
Initialize I = 0
20  Read A(I)
    Read B(I)
    Store C(I) = A(I) + B(I)
    Increment I = i + 1
    If I ≤ 100 goto 20
```

### Vector computer

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

# VECTOR INSTRUCTIONS

f1: $V * V$	V: Vector operand S: Scalar operand
f2: $V * S$	
f3: $V \times V * V$	
f4: $V \times S * V$	

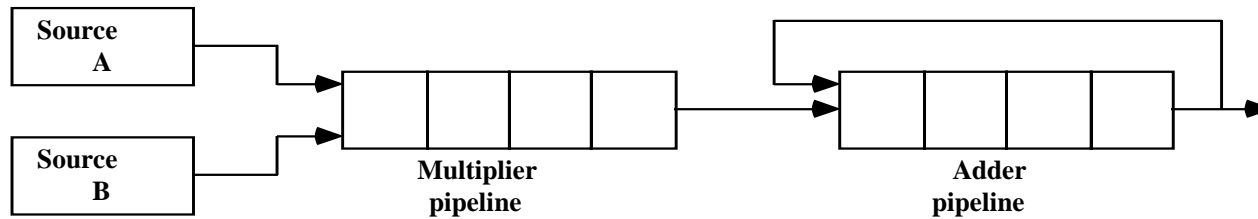
Type	Mnemonic	Description (l = 1, ..., n)	
f1	VSQR	Vector square root	$B(l) * \text{SQR}(A(l))$
	VSIN	Vector sine	$B(l) * \sin(A(l))$
	VCOM	Vector complement	$A(l) * \overline{A(l)}$
f2	VSUM	Vector summation	$S * \Sigma A(l)$
	VMAX	Vector maximum	$S * \max\{A(l)\}$
f3	VADD	Vector add	$C(l) * A(l) + B(l)$
	VMPY	Vector multiply	$C(l) * A(l) * B(l)$
	VAND	Vector AND	$C(l) * A(l) . B(l)$
	VLAR	Vector larger	$C(l) * \max(A(l), B(l))$
	VTGE	Vector test >	$C(l) * 0$ if $A(l) < B(l)$ $C(l) * 1$ if $A(l) > B(l)$
f4	SADD	Vector-scalar add	$B(l) * S + A(l)$
	SDIV	Vector-scalar divide	$B(l) * A(l) / S$

# VECTOR INSTRUCTION FORMAT

## Vector Instruction Format

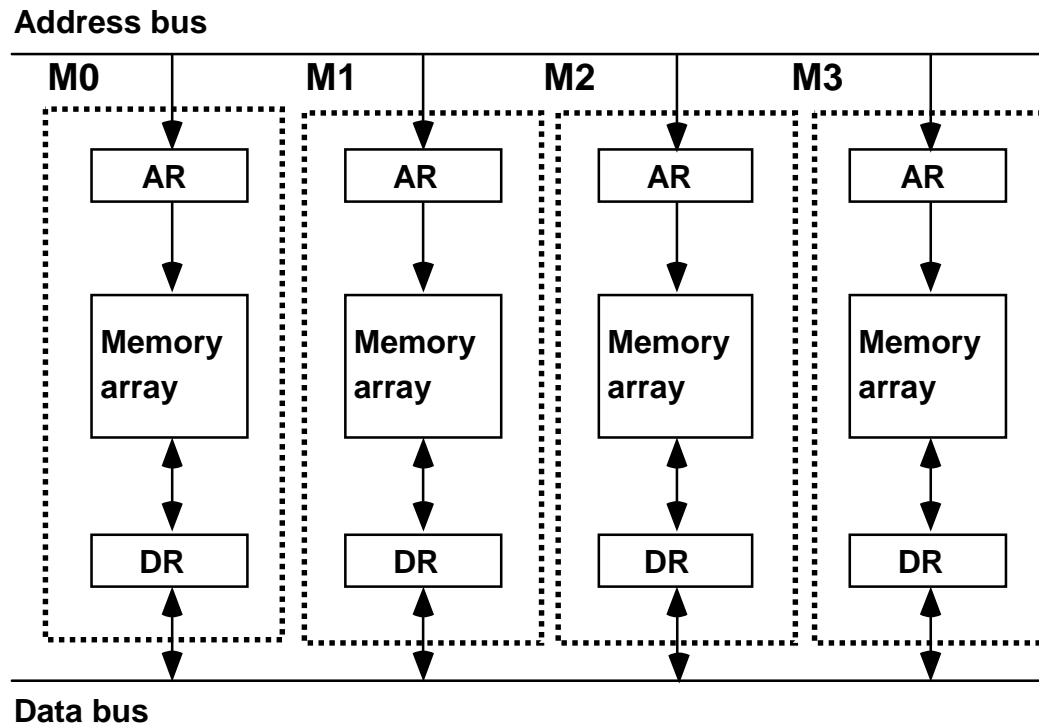
Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
----------------	-----------------------	-----------------------	--------------------------	---------------

## Pipeline for Inner Product



# MULTIPLE MEMORY MODULE AND INTERLEAVING

## Multiple Module Memory



## Address Interleaving

Different sets of addresses are assigned to different memory modules