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

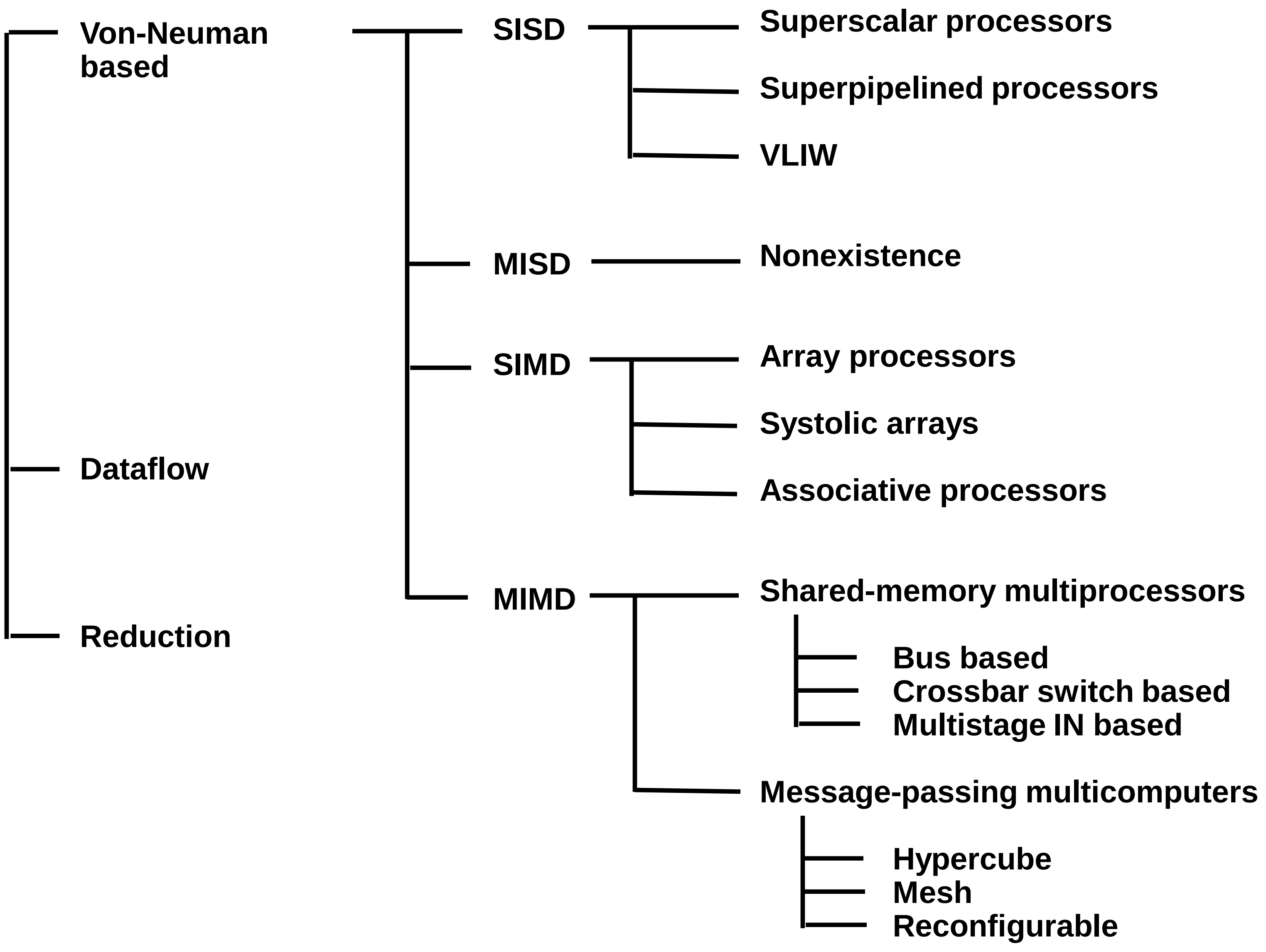
Architectural Classification

– **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**

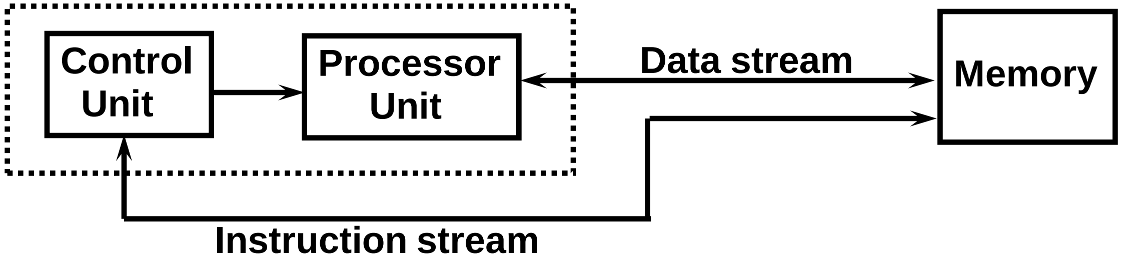
		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	SISD	SIMD
	Multiple	MISD	MIMD

COMPUTER ARCHITECTURES FOR PARALLEL PROCESSING



- Von-Neuman based
  - SISD
    - Superscalar processors
    - Superpipelined processors
    - VLIW
  - MISD
    - Nonexistence
  - SIMD
    - Array processors
    - Systolic arrays
    - Associative processors
  - MIMD
    - Shared-memory multiprocessors
      - Bus based
      - Crossbar switch based
      - Multistage IN based
    - Message-passing multicomputers
      - Hypercube
      - Mesh
      - Reconfigurable
- Dataflow
- Reduction

SISD COMPUTER SYSTEMS



Characteristics

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

Limitations

Von Neumann bottleneck

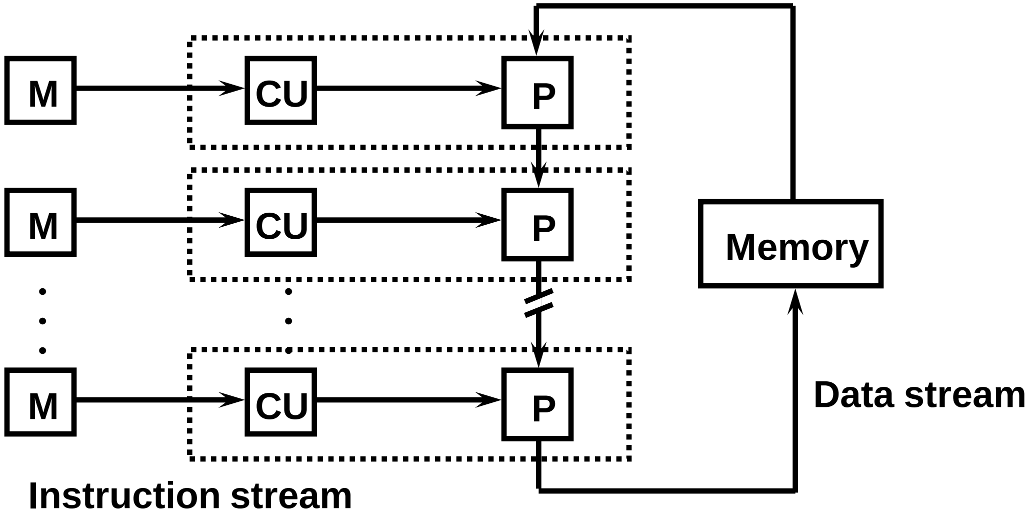
Maximum speed of the system is limited by the *Memory Bandwidth* (bits/sec or bytes/sec)

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

SISD PERFORMANCE IMPROVEMENTS

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

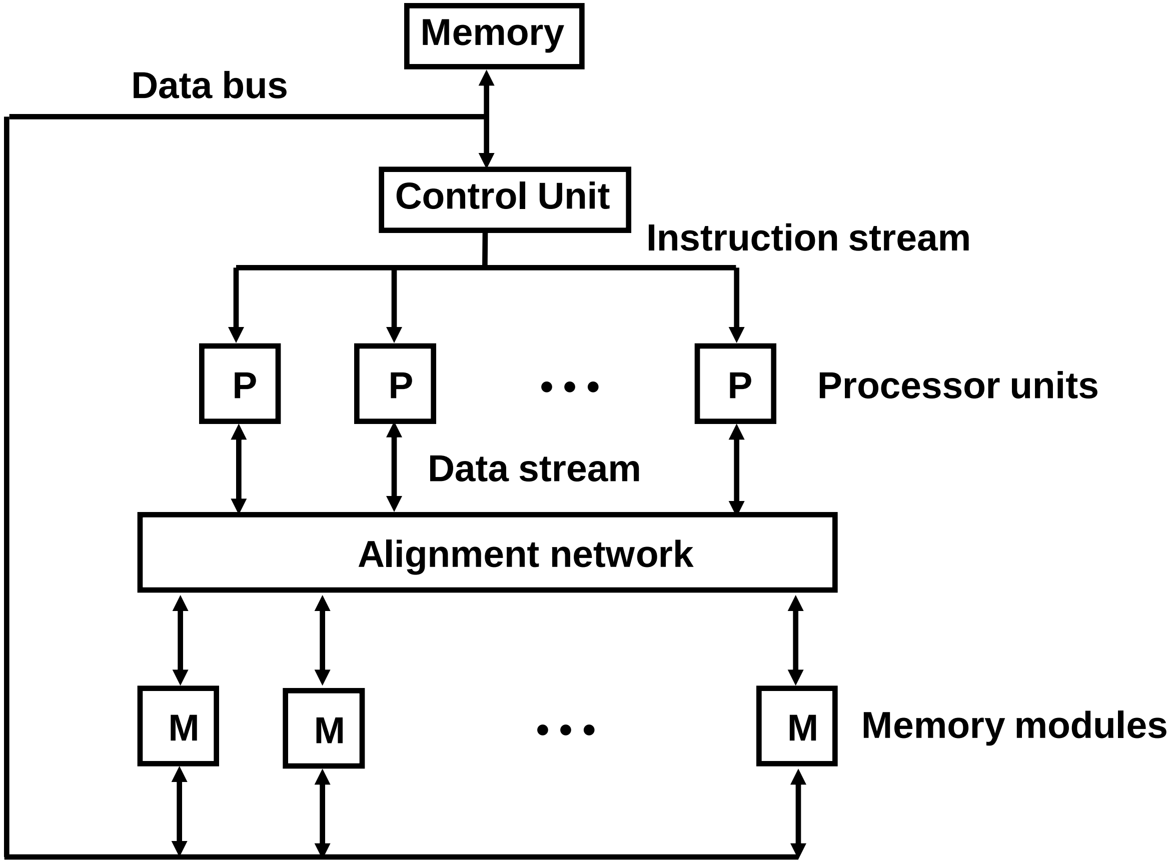
MISD COMPUTER SYSTEMS



Characteristics

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

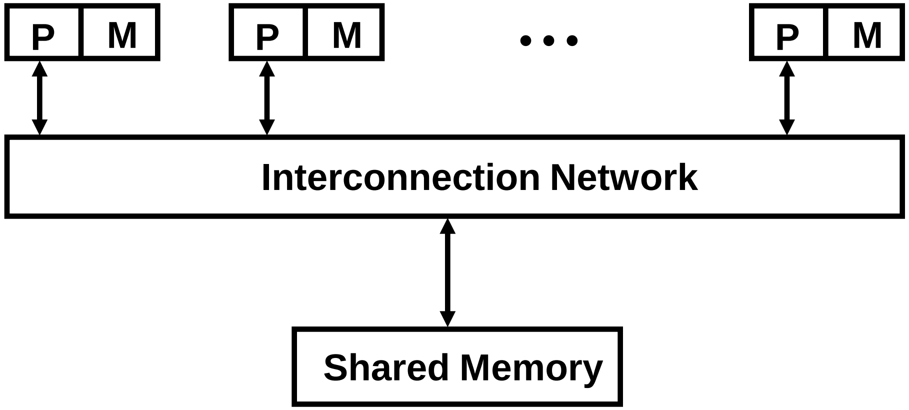
SIMD COMPUTER SYSTEMS



Characteristics

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

MIMD COMPUTER SYSTEMS



Characteristics

- Multiple processing units

- Execution of multiple instructions on multiple data

Types of MIMD computer systems

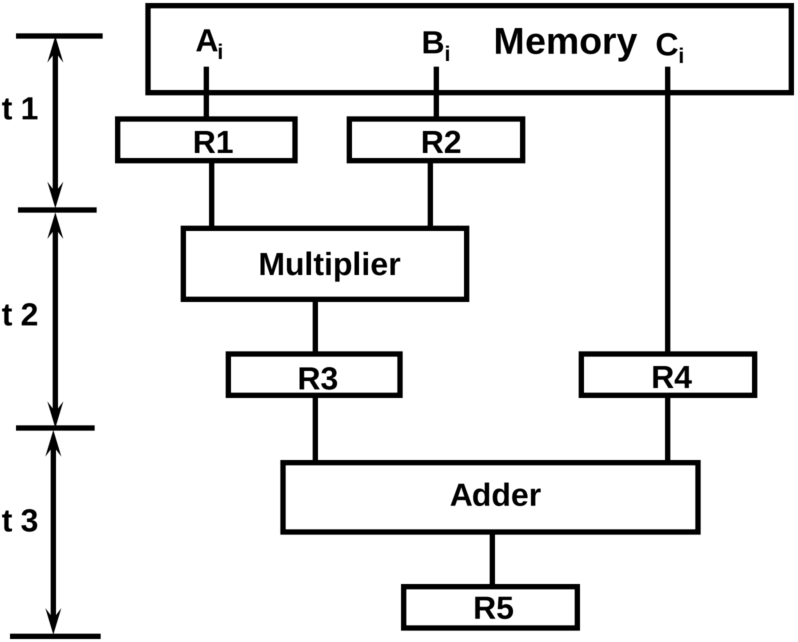
- Shared memory multiprocessors

- Message-passing multicomputers

PIPELINING

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

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



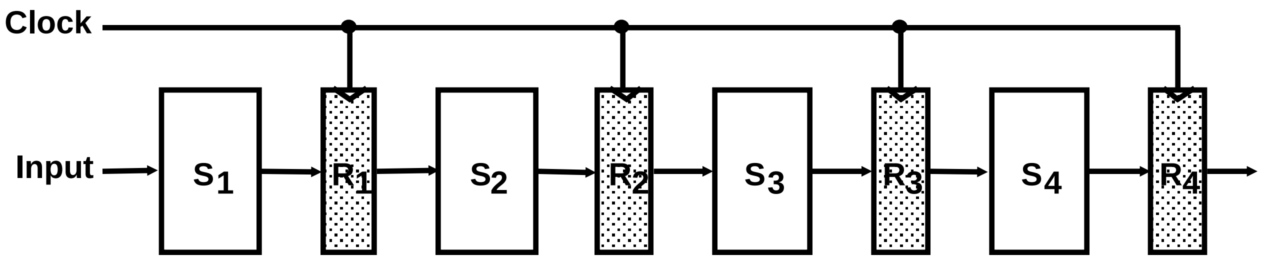
R1 ← A_i, R2 ← B_i	**Load A_i and B_i**
R3 ← R1 * R2, R4 ← C_i	**Multiply and load C_i**
R5 ← R3 + R4	**Add**

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

Clock cycles

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

PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

τ_1: Time required to complete the n tasks

$\tau_1 = n * t_n$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

τ_κ: Time required to complete the n tasks

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

Speedup

S_k: Speedup

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

$$\lim_{n \to \infty} S_k = \frac{t_n}{t_p}$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

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

Pipelined System

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

Non-Pipelined System

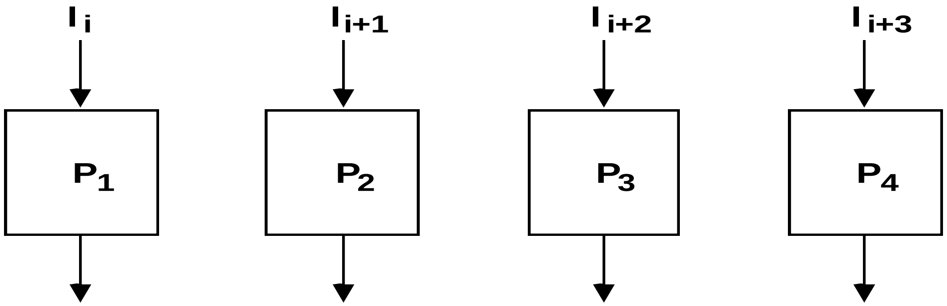
$n*k*t_p = 100 * 80 = 8000$nS

Speedup

$S_k = 8000 / 2060 = 3.88$

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

Multiple Functional Units

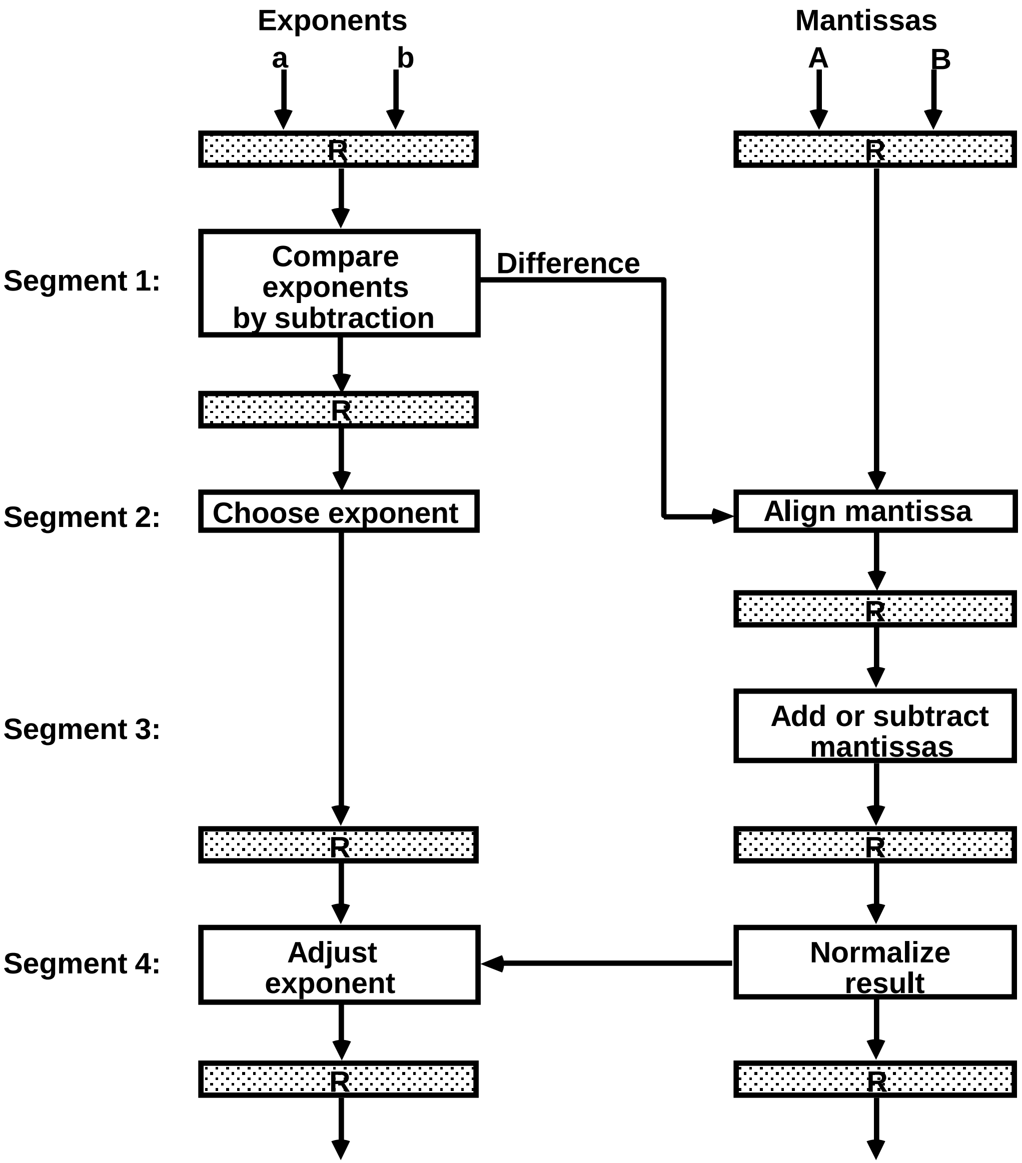


ARITHMETIC PIPELINE

Floating-point adder

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

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



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

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

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

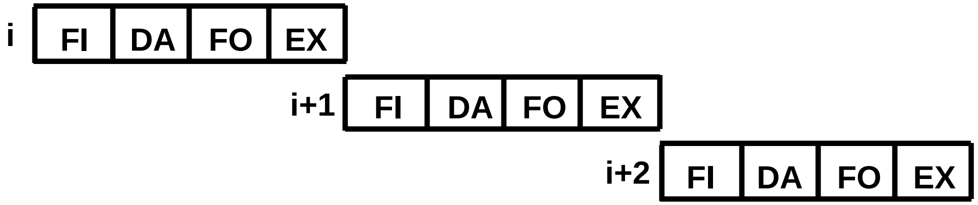
==> 4-Stage Pipeline

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

INSTRUCTION PIPELINE

Execution of Three Instructions in a 4-Stage Pipeline

Conventional

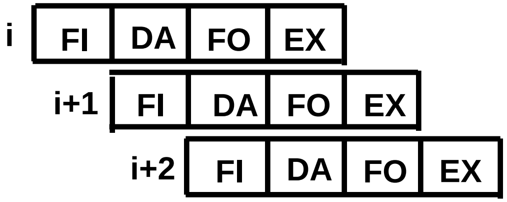


i FI DA FO EX

i+1 FI DA FO EX

i+2 FI DA FO EX

Pipelined

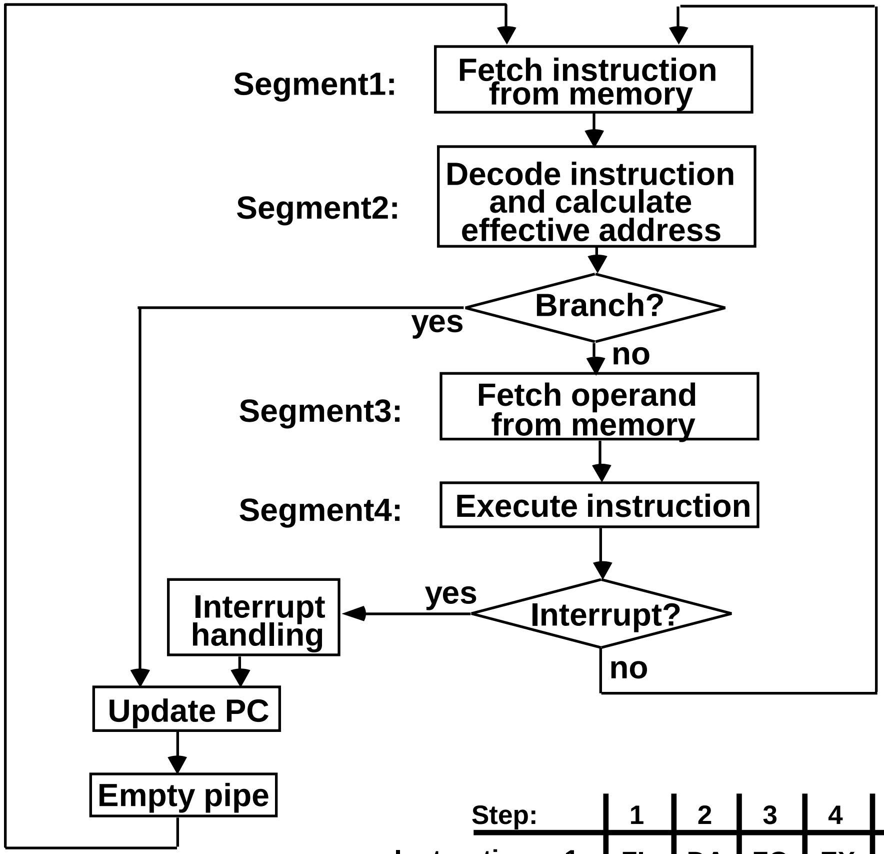


i FI DA FO EX

i+1 FI DA FO EX

i+2 FI DA FO EX

INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



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

MAJOR HAZARDS IN PIPELINED EXECUTION

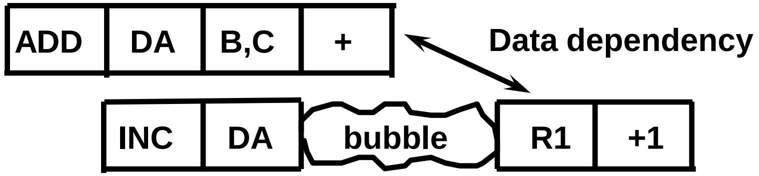
Structural hazards(Resource Conflicts)

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

Data hazards (Data Dependency Conflicts)

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

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



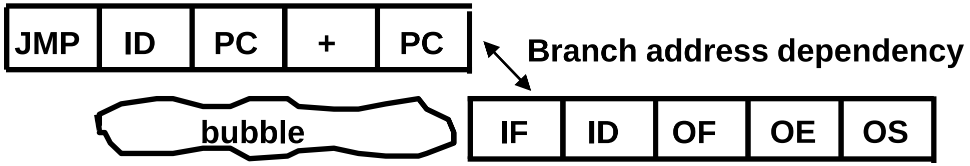
ADD | DA | B,C | +

INC | DA | bubble | R1 | +1

Data dependency

Control hazards

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



JMP | ID | PC | + | PC

bubble | IF | ID | OF | OE | OS

Branch address dependency

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

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	E	
Store R3				I	A	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
f2: V ∗ S
f3: V x V ∗ V
f4: V x S ∗ V

V: Vector operand
S: Scalar operand

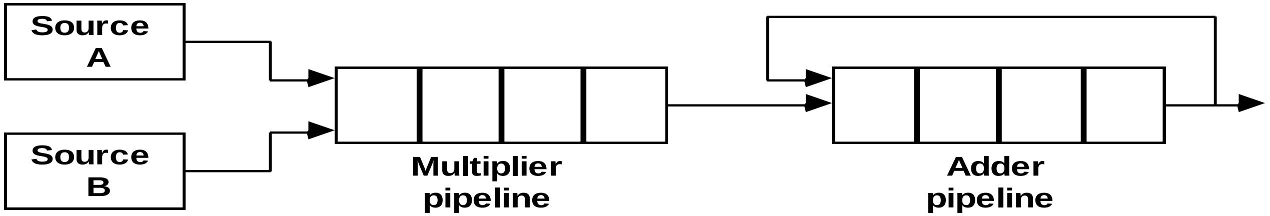
Type	Mnemonic	Description (I = 1, ..., n)	
f1	VSQR	Vector square root	B(I) ∗ SQR(A(I))
	VSIN	Vector sine	B(I) ∗ sin(A(I))
	VCOM	Vector complement	A(I) ∗ $\overline{A(I)}$
f2	VSUM	Vector summation	S ∗ Σ A(I)
	VMAX	Vector maximum	S ∗ max{A(I)}
f3	VADD	Vector add	C(I) ∗ A(I) + B(I)
	VMPY	Vector multiply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

VECTOR INSTRUCTION FORMAT

Vector Instruction Format

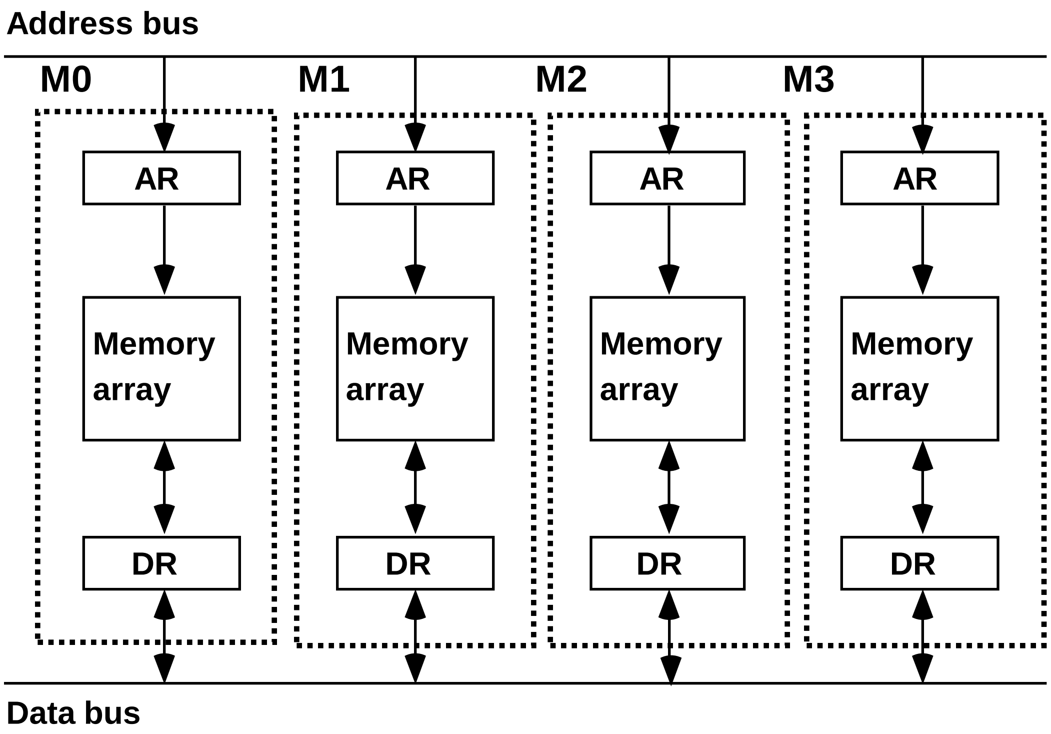
Operation code	Base address source 1	Base address source 2	Base address destination	Vector length

Pipeline for Inner Product



MULTIPLE MEMORY MODULE AND INTERLEAVING

Multiple Module Memory



Address Interleaving

Different sets of addresses are assigned to different memory modules