Unit 6 :Pipeline and Vector Processing

Parallel computers

Parallel computers are those that emphasize the parallel processing between the operations in some way. In the previous unit, all the basic terms of parallel processing and computation have been defined. Parallel computers can be characterized based on the data and instruction streams forming various types of computer organizations. They can also be classified based on the computer structure, e.g. multiple processors having separate memory or one shared global memory. Parallel processing levels can also be defined based on the size of instructions in a program called grain size. Thus, parallel computers can be classified based on various criteria. This unit discusses all types of classification of parallel computers based on the above mentioned criteria.

TYPES OF CLASSIFICATION

The following classification of parallel computers have been identified:

1) Classification based on the instruction and data streams
2) Classification based on the structure of computers
3) Classification based on how the memory is accessed.
4) Classification based on grain size .

Qn What is Parallel Computer? explain

Parallel Processing

- Parallel processing – denotes the use of techniques designed to perform various data processing tasks simultaneously to increase a computer's overall speed.
- These techniques can include:
  - performing arithmetic or logical operations while fetching the next instruction
  - executing several instructions at the same time
  - performing arithmetic or logical operations on multiple sets of operands.
- While parallel processing can be more expensive, technological advances have dropped to overall cost of processor design enough to make it financially feasible.

Levels of Complexity in Parallel Processing

- On the low level:
  - Shift registers are sequential; parallel load registers operate all their bits simultaneously.
- On the high level:
  - Multiple functional units allow all multiple operations to be executed concurrently.

Multiple functional Units of Parallel Processing

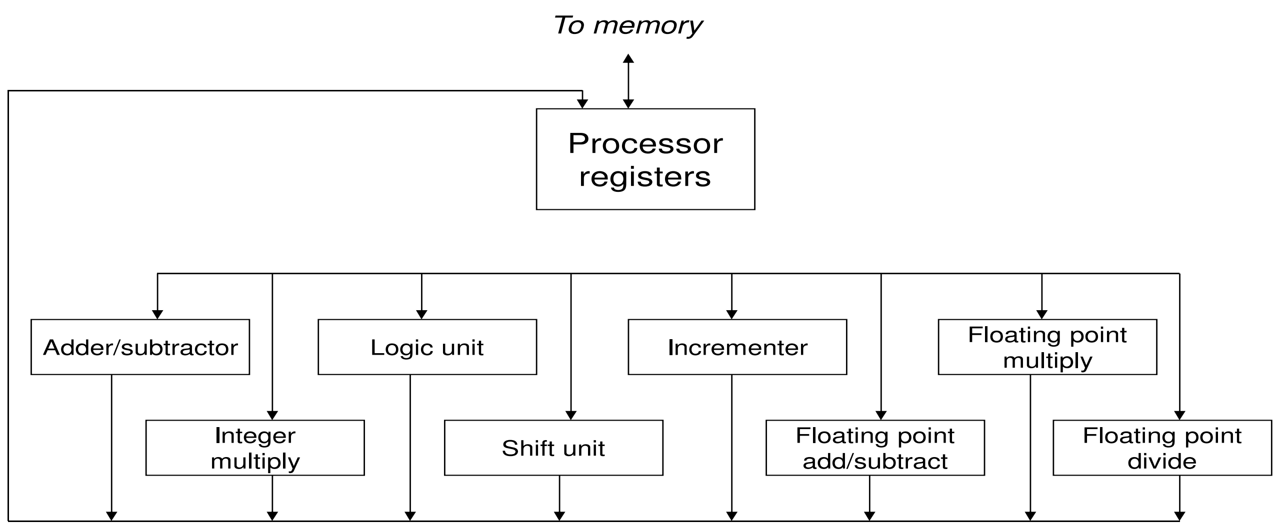
Instead of processing each instruction sequentially, a parallel processing system provides concurrent data processing to increase the execution time.

In this the system may have two or more ALU's and should be able to execute two or more instructions at the same time. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput.

NOTE: Throughput is the number of instructions that can be executed in a unit of time.

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. At the higher level of complexity, parallel processing can be achieved by using multiple functional units that perform many operations simultaneously.

Processor With Multiple Functional Units



What do you mean by parallel computing ?Expain parallel computing with its multiple functional units?

Flynn classification:

It is based on multiplicity of instruction streams and the data streams in computer systems.

This classification was first studied and proposed by Michael Flynn in 1972. Flynn did not consider the machine architecture for classification of parallel computers; he introduced the concept of instruction and data streams for categorizing of computers. All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn's classification. Since, this classification is based on instruction and data streams, first we need to understand how the instruction cycle works.

Instruction Cycle

The instruction cycle consists of a sequence of steps needed for the execution of an instruction in a program. A typical instruction in a program is composed of two parts: Opcode and Operand. The Operand part specifies the data on which the specified operation is to be done. (See Figure 1). The Operand part is divided into two parts: addressing mode and the Operand. The addressing mode specifies the method of determining the addresses of the actual data on which the operation is to be performed and the operand part is used as an argument by the method in determining the actual address.

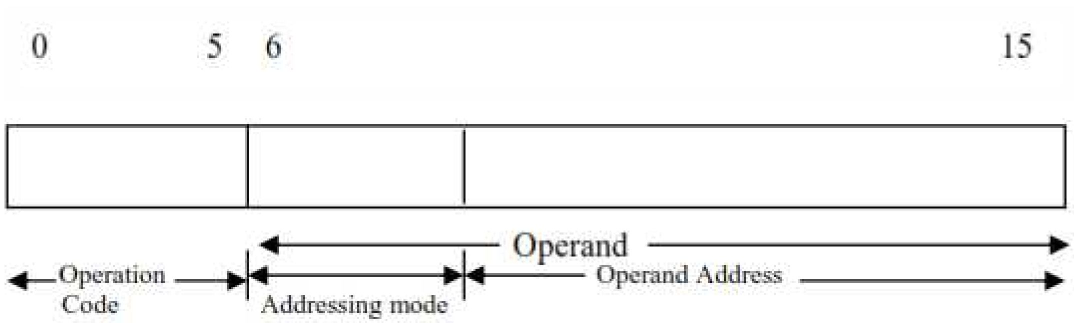


Figure 1: Opcode and Operand

The control unit of the CPU of the computer fetches instructions in the program, one at a time. The fetched Instruction is then decoded by the decoder which is a part of the control unit and the processor executes the decoded instructions. The result of execution is temporarily stored in Memory Buffer Register (MBR) (also called Memory Data Register). The normal execution steps are shown in Figure 2.

Qn Explain the Flynn classification of parallel computing?

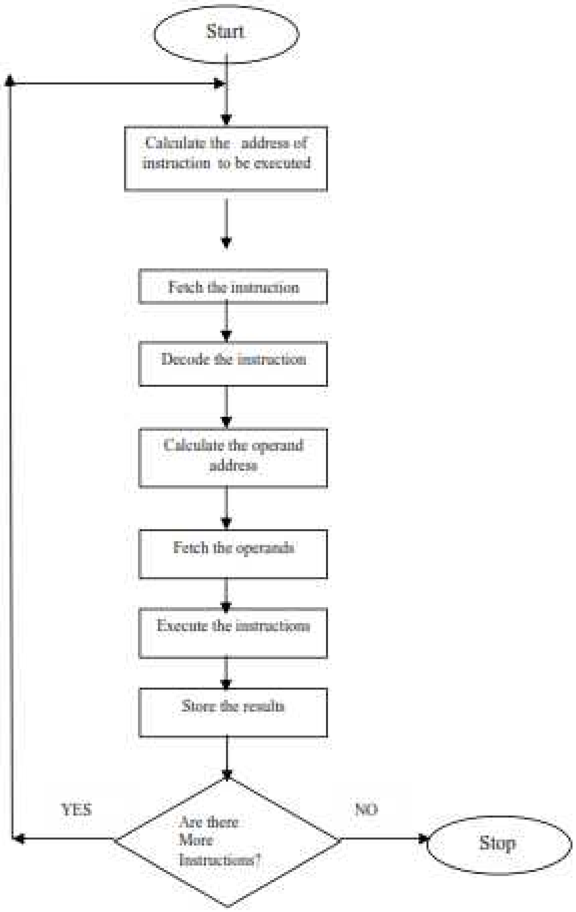


Figure 2: Instruction execution steps

Instruction Stream and Data Stream

The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called instruction stream. Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called data stream. These two types of streams are shown in Figure 3.

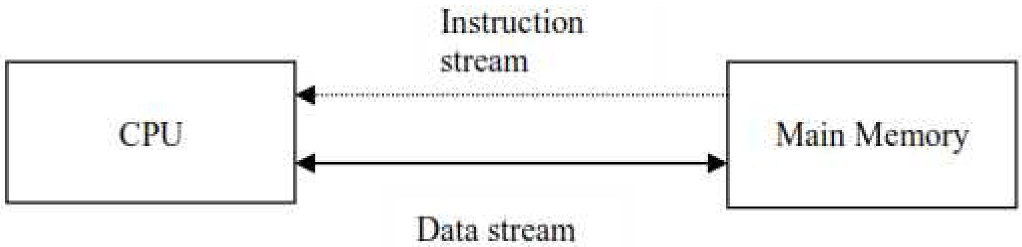
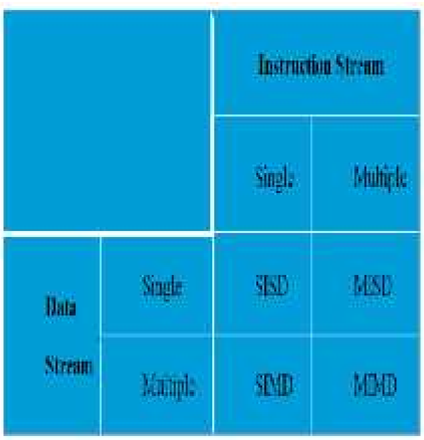


Figure 3: Instruction and data stream

Thus, it can be said that the sequence of instructions executed by CPU forms the Instruction streams and sequence of data (operands) required for execution of instructions form the Data streams.

Flynn's Classification



Flynn's classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let Is and Ds are minimum number of streams flowing at any point in the execution, then the computer organisation can be categorized as follows:

1) Single Instruction and Single Data stream (SISD)

In this organisation, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit as shown in Figure 4. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organisation is depicted in the diagram: Is = Ds = 1

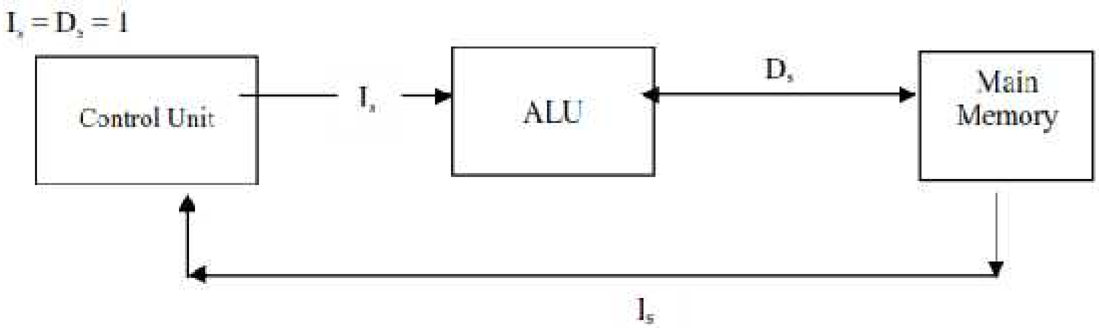


Figure 4: SISD Organisation

Examples of SISD machines include:

- CDC 6600 which is unpipelined but has multiple functional units.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing.

Captions: CU - Control Unit ; PU – Processing Unit MU – Memory Unit ; IS – Instruction Stream DS – Date Stream

2) Single Instruction and Multiple Data stream (SIMD):

In this organization, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory as shown in Figure 5. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous. Examples of SIMD organisation are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).

This type of computer organisation is denoted as:

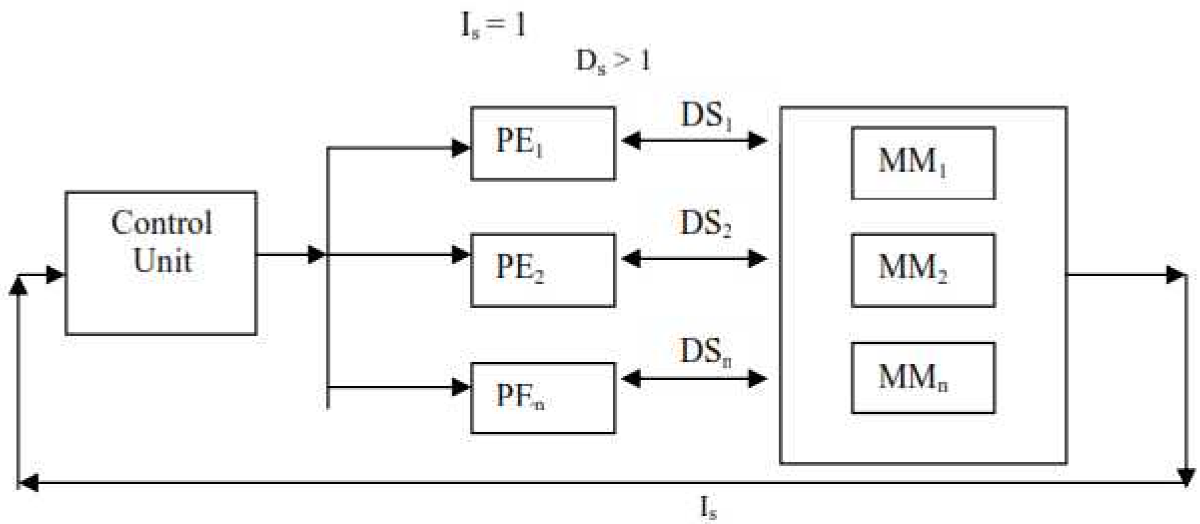


Figure 5: SIMD Organisation

Captions: CU - Control Unit ; PU - Processing Unit MU - Memory Unit ; IS - Instruction Stream DS - Date Stream ; PE – Processing Element LM – Local Memory

3) Multiple Instruction and Single Data stream (MISD)

In this organization, multiple processing elements are organised under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification. All processing elements are interacting with the common shared memory for the organisation of single data stream as shown in Figure 6. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University. This type of computer organisation is denoted as:

Is > 1

Ds = 1

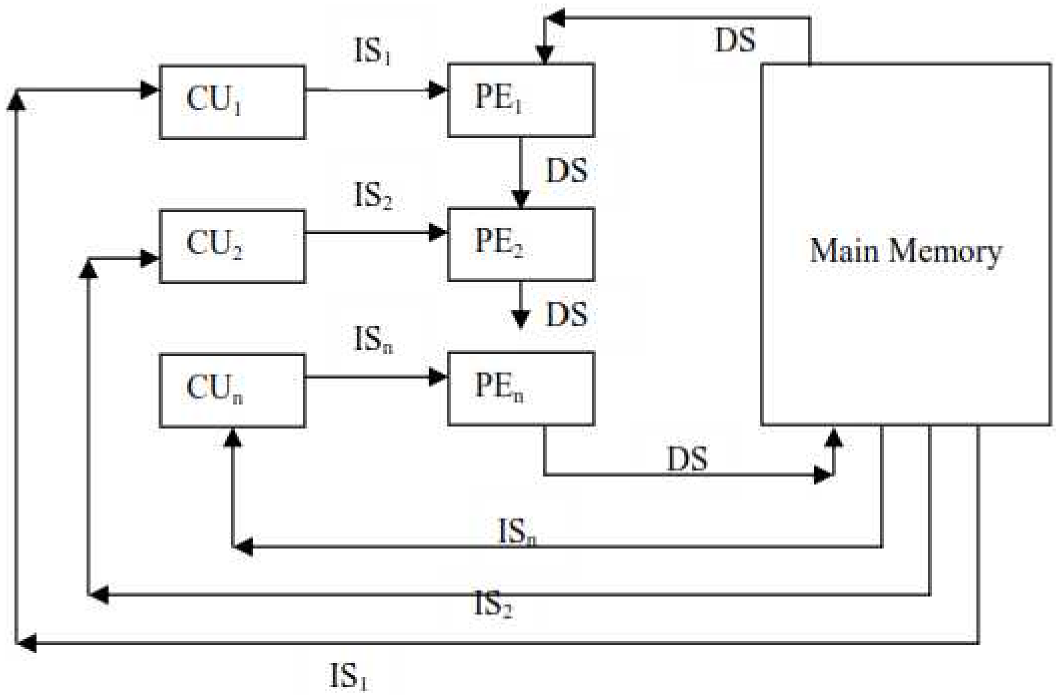


Figure 6: MISD Organisation

This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied. But for the specialized applications, MISD organisation can be very helpful. For example, Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data. This is also known as N- version programming. All these redundant data are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

4) Multiple Instruction and Multiple Data stream (MIMD)

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams . Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory as shown in Figure 7. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organization is said to be a Parallel computer. All multiprocessor systems fall under this classification.

Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m*, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC. This type of computer organisation is denoted as:

Is > 1 Ds > 1

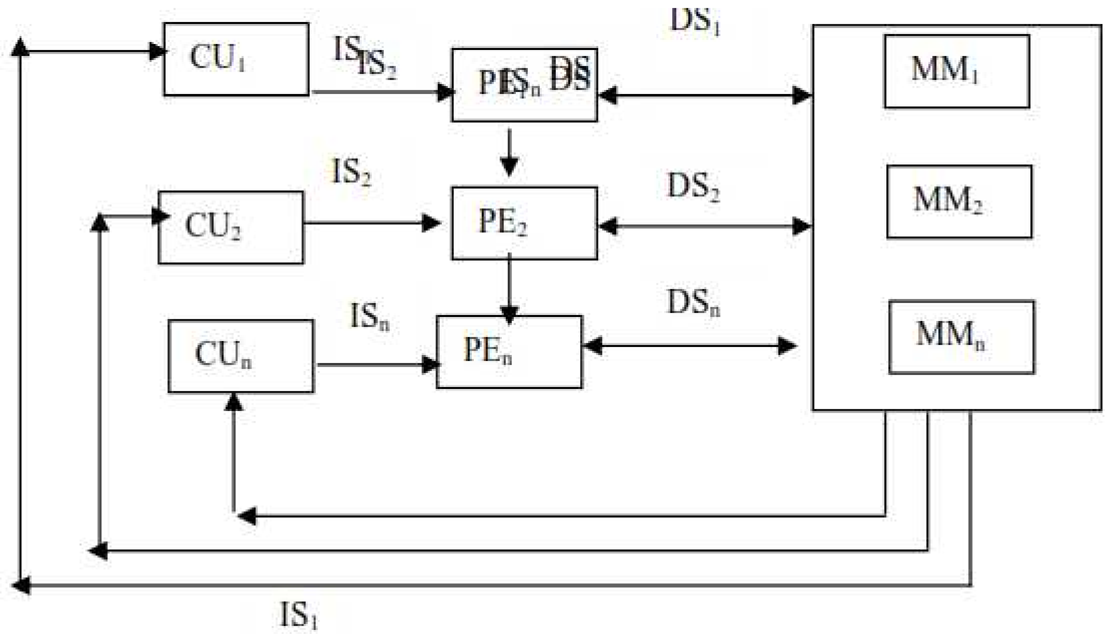


Figure 7: MIMD Organisation

Of the classifications discussed above, MIMD organization is the most popular for a parallel computer. In the real sense, parallel computers execute the instructions in MIMD mode.

CU - Control Unit ; PU - Processing Unit MU - Memory Unit ; IS - Instruction Stream

DS - Date Stream ; PE – Processing Element LM – Local Memory

STRUCTURAL CLASSIFICATION

Flynn's classification discusses the behavioural concept and does not take into consideration the computer's structure. Parallel computers can be classified based on their structure also, which is discussed below and shown in Figure 8.

As we have seen, a parallel computer (MIMD) can be characterised as a set of multiple processors and shared memory or memory modules communicating via an interconnection network. When multiprocessors communicate through the global shared memory modules then this organisation is called Shared memory computer or Tightly coupled systems as shown in Figure 9. Similarly when every processor in a multiprocessor system, has its own local memory and the processors communicate via messages transmitted between their local memories, then this organisation is called Distributed memory computer or Loosely coupled system as shown in Figure 10. Figures 9 and 10 show the simplified diagrams of both organisations.

The processors and memory in both organisations are interconnected via an interconnection network. This interconnection network may be in different forms like crossbar switch, multistage network, etc. which will be discussed in the next unit.

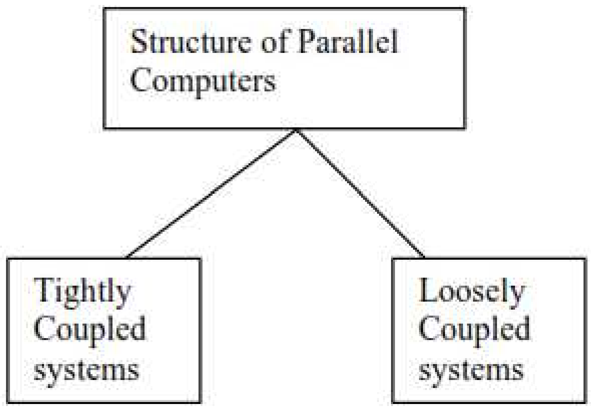


Figure 8: Structural classification

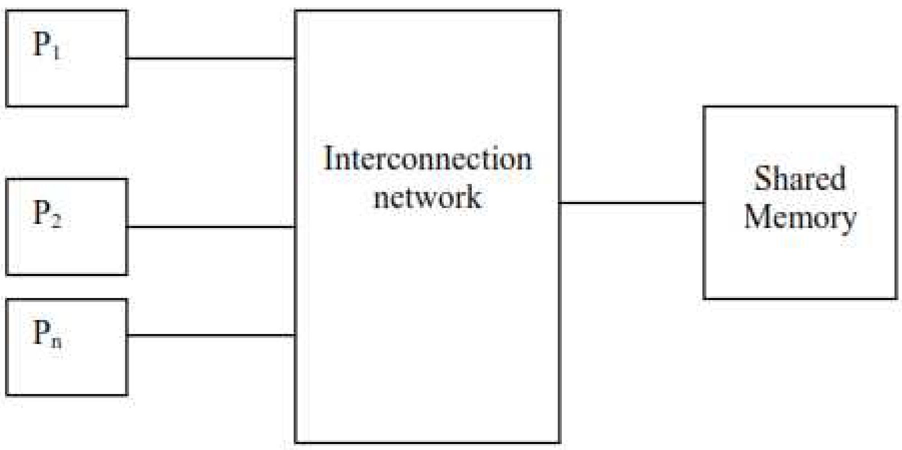


Figure 9: Tightly coupled system

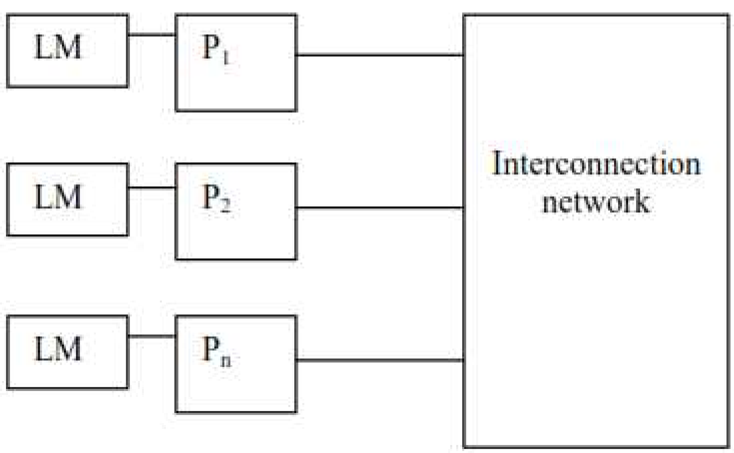


Figure 10 Loosely coupled system

2.5.1 Shared Memory System / Tightly Coupled System

Shared memory multiprocessors have the following characteristics:

- Every processor communicates through a shared global memory.

- For high speed real time processing, these systems are preferable as their throughput is high as compared to loosely coupled systems.

In tightly coupled system organization, multiple processors share a global main memory, which may have many modules as shown in detailed Figure 11. The processors have also access to I/O devices. The inter- communication between processors, memory, and other devices are implemented through various interconnection networks, which are discussed below.

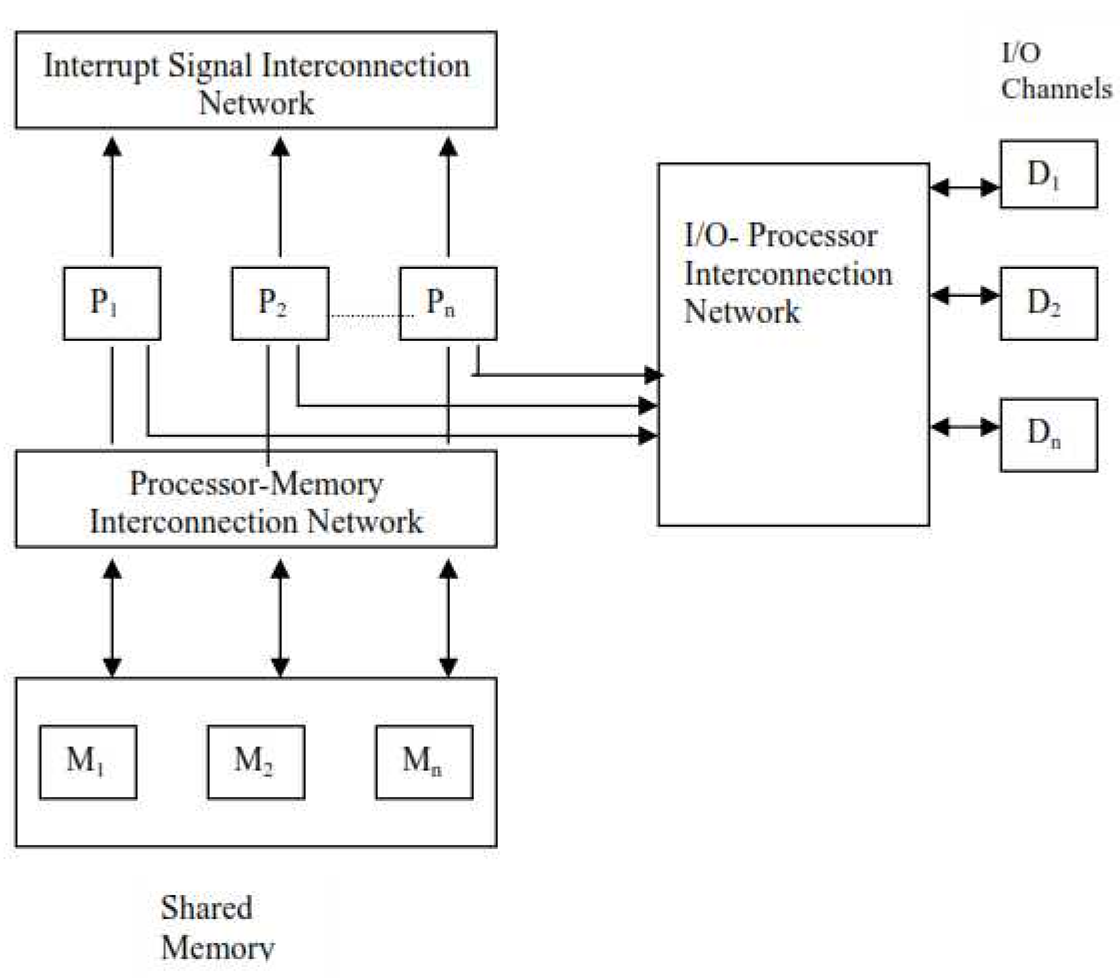


Figure 11: Tightly coupled system organization

i) Processor-Memory Interconnection Network (PMIN)

This is a switch that connects various processors to different memory modules. Connecting every processor to every memory module in a single stage while the crossbar switch may become complex. Therefore, multistage network can be adopted. There can be a conflict among processors such that they attempt to access the same memory modules. This conflict is also resolved by PMIN.

ii) Input-Output-Processor Interconnection Network (IOPIN)

This interconnection network is used for communication between processors and I/O channels. All processors communicate with an I/O channel to interact with an I/O device with the prior permission of IOPIN.

iii) Interrupt Signal Interconnection Network (ISIN)

When a processor wants to send an interruption to another processor, then this interrupt first goes to ISIN, through which it is passed to the destination processor. In this way, synchronisation between processor is implemented by ISIN. Moreover, in case of failure of one processor, ISIN can broadcast the message to other processors about its failure.

Since, every reference to the memory in tightly coupled systems is via interconnection network, there is a delay in executing the instructions. To reduce this delay, every processor may use cache memory for the frequent references made by the processor as shown in Figure 12.

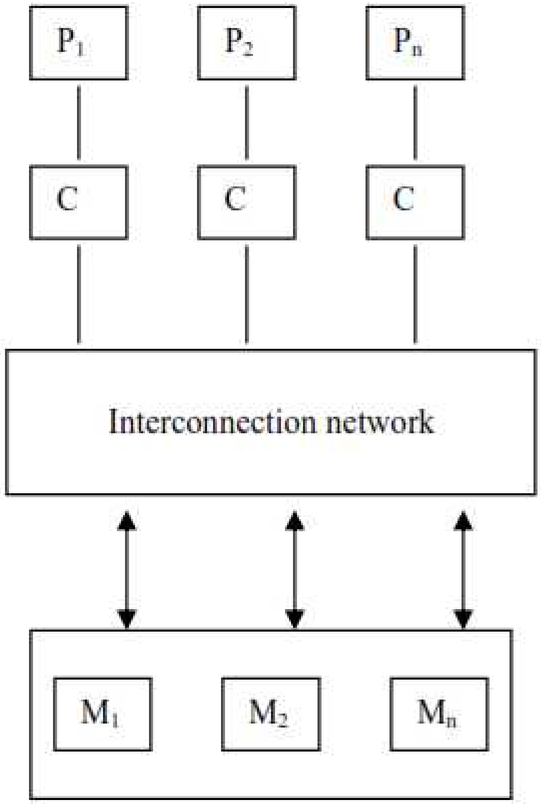


Figure 12: Tightly coupled systems with cache memory

The shared memory multiprocessor systems can further be divided into three modes which are based on the manner in which shared memory is accessed. These modes are shown in Figure 13 and are discussed below.

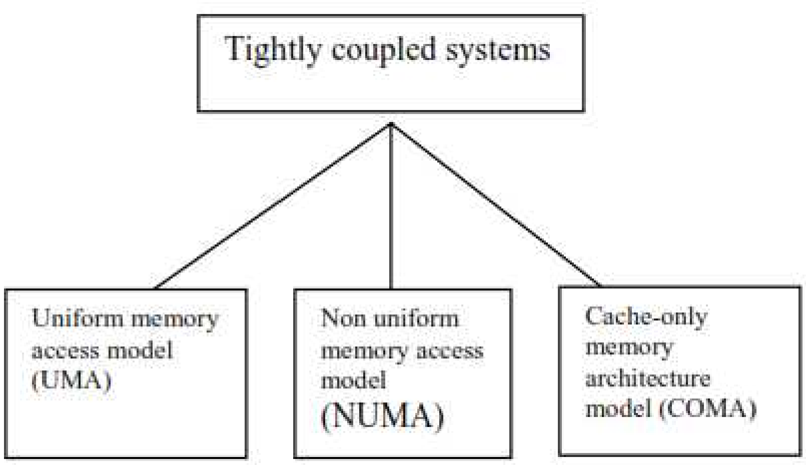


Figure 13: Modes of Tightly coupled systems

2.5.1.1 Uniform Memory Access Model (UMA)

In this model, main memory is uniformly shared by all processors in multiprocessor systems and each processor has equal access time to shared memory. This model is used for time-sharing applications in a multi user environment.

2.5.1.2 Non-Uniform Memory Access Model (NUMA)

In shared memory multiprocessor systems, local memories can be connected with every processor. The collection of all local memories form the global memory being shared. In this way, global memory is distributed to all the processors. In this case, the access to a local memory is uniform for its corresponding processor as it is attached to the local memory. But if one reference is to the local memory of some other remote processor, then the access is not uniform. It depends on the location of the memory. Thus, all memory words are not accessed uniformly.

2.5.1.3 Cache-Only Memory Access Model (COMA)

As we have discussed earlier, shared memory multiprocessor systems may use cache memories with every processor for reducing the execution time of an instruction. Thus in NUMA model, if we use cache memories instead of local memories, then it becomes COMA model. The collection of cache memories form a global memory space. The remote cache access is also non-uniform in this model.

2.5.2 Loosely Coupled Systems

These systems do not share the global memory because shared memory concept gives rise to the problem of memory conflicts, which in turn slows down the execution of instructions. Therefore, to alleviate this problem, each processor in loosely coupled systems is having a large local memory (LM), which is not shared by any other processor. Thus, such systems have multiple processors with their own local memory and a set of I/O devices. This set of processor, memory and I/O devices makes a computer system. Therefore, these systems are also called multi-computer systems. These computer systems are connected together via message passing interconnection network through which processes communicate by passing messages to one another. Since every computer system or node in multicomputer systems has a separate memory, they are called distributed multicomputer systems. These are also called loosely coupled systems, meaning that nodes have little coupling between them as shown in Figure 14.

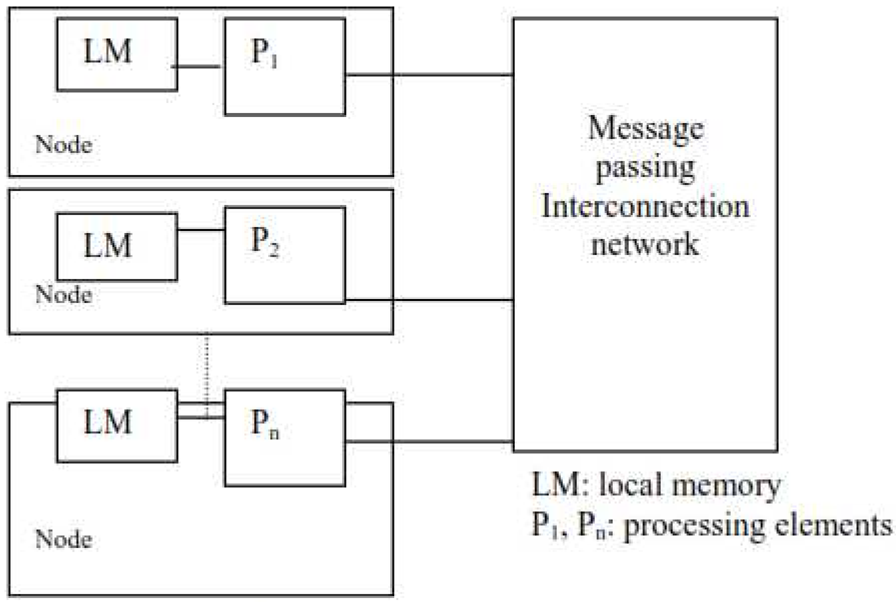


Figure 14: Loosely coupled system organisation

Since local memories are accessible to the attached processor only, no processor can access remote memory. Therefore, these systems are also known as no-remote memory access (NORMA) systems. Message passing interconnection network provides connection to every node and inter-node communication with message depends on the type of interconnection network. For example, interconnection network for a non-hierarchical system can be shared bus.

Flynn's Taxonomy

- Michael Flynn classified computers according to their type of parallelism:
  - **SISD** – Single Instruction Single Data – simple computers that are essentially devoid of parallelism
  - **SIMD** – Single Instruction Multiple Data – processors capable of performing the same operation on multiple pairs of operands
  - **MISD** – Multiple Instruction Single Data – performing several operations on the same set of data – only of theoretical interest
  - **MIMD** - Multiple Instruction Multiple Data – capable of processing several programs simultaneously on different sets of data

Pipelining

- Pipelining is a technique where sequential processes are broken down into separate suboperations, each of which being performed by its own hardware.
- Each computation is passed along to the next segment in the pipeline, with the processes are carried in a manner analogous to an assembly line.
- The fact that each suboperation is performed by different hardware allows different stages of the overall operation to be performed in parallel.

Picturing The Pipeline

- It may be easier to picture a segment of the pipeline as a register followed by a combinational circuit.
  - The register holds the data
  - The combinational circuit performs the specified operation.
  - The result is passed on to another register.
- The suboperations are synchronized by the clock pulses.

Pipelining: An Example

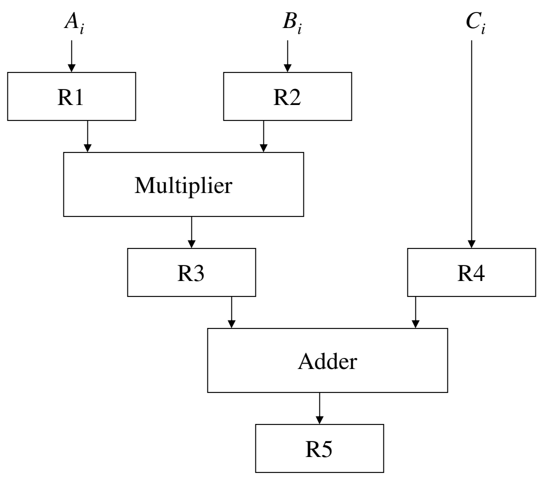
- Imagine that we want to evaluate the following expression for seven sets of values:

  $A_i*B_i + C_i$, for $i = 1, 2, 3, \ldots, 7$
- Each suboperation can be implemented by a different segment within the pipeline.
- This can be decomposed into three segments:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	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 the product

- The 5 registers are each loaded on a new clock pulse.

Pipeline Processing



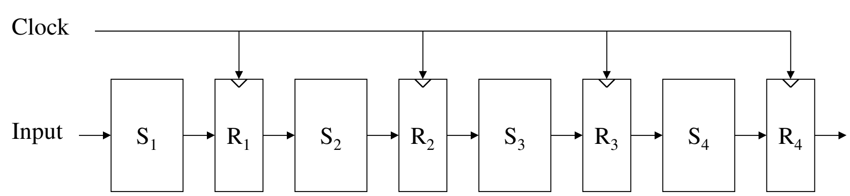
Registers in the Pipeline

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

Pipelining – General Considerations

- Any operation that can be decomposed into a series of suboperations of the same complexity can be implemented by pipelining.
- We define a task as the total operation that performing when going through the entire pipeline.

4-Segment Pipeline



Space-Time Diagram

Segment:

1	2	3	4	5	6	7	8	9
T_1	T_2	T_3	T_4	T_5	T_6			
	T_1	T_2	T_3	T_4	T_5	T_6		
		T_1	T_2	T_3	T_4	T_5	T_6	
			T_1	T_2	T_3	T_4	T_5	T_6

Clock cycles

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.

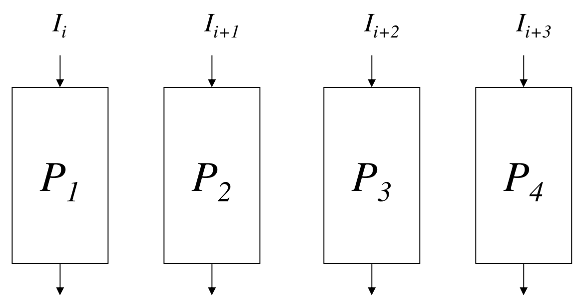
Speedup – An Example

- Let's assume that
  - $t_p = 20$ns
  - the pipeline has $k = 4$ segments
  - executes $n = 100$ tasks in sequence
- The pipeline will

  $(k - n + 1)\ t_p \quad = (4 + 100 - 1) \times 20$ ns
  $= 2060$ ns to complete the task
- Assuming tn = $4 \times 20 = 80$ns, it will require

  $kt_p = 100 \times 80 = 8000$ ns
- Therefore the speedup is 8000/2060 = **3.88** which will approach 4 as n grows

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.

Example – Floating Point Addition

- The operands in floating point addition are 2 normalized binary numbers:

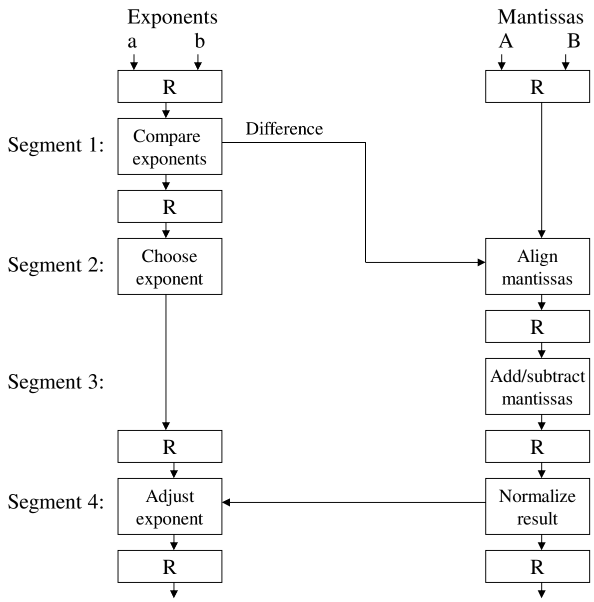
  $X = A \times 2^a$

  $Y = B \times 2^b$

  where X and y are fractions representing the mantissa

  a and b are the exponents
- The four segments are:
  1. Compare the exponents
  2. Align the mantissas
  3. Add or subtract the mantissas
  4. Normalize the results

Pipeline for Floating Point Addition/Subtraction



Example – Floating Point Addition

- If our operands are
  $X = 0.9504 \times 10^3$
  $Y = 0.8200 \times 10^2$
- The difference of the exponents is 3-2 = 1; we adjust Y's exponent:
  $X = 0.9504 \times 10^3$
  $Y = 0.0820 \times 10^3$
- We calculate the product:
  $Z = 1.0324 \times 10^3$
- Then we normalize:
  $Z = 0.10324 \times 10^4$

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's 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

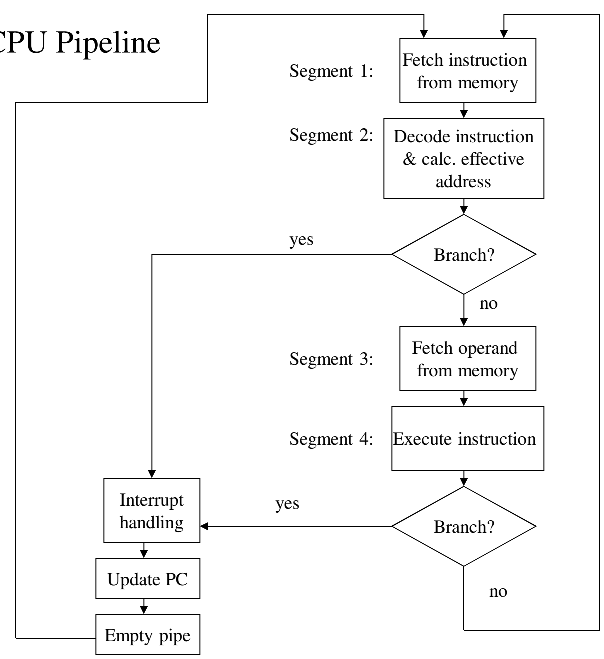
Difficulties Slowing Down the Instruction Pipeline

- 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 resolve 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)
- While one instruction is being executed in stage 4, another is fetching an operand in stage 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 stages, delete the instructions in the instruction buffer and restart the pipeline with the new address.

4-Segment CPU Pipeline



Timing of Instruction Pipeline

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

Decoding a branch instruction

Everything has to wait until the branch has finished executing

FI – Fetch instruction
DA – Decode instruction and calculate eff. address
FO – Fetch operand
EX – Execute instruction

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 that change the PC's value.

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 first 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 suboperations for several reasons:
  - Its fixed length format means that decoding can occur during register selection
  - Data manipulation are all done using register-to-register operations, so there is no need to calculate effective addresses.
  - This means that instructions can be carried out in 3 subsoperations, with the third used for storing the result in the specified register.

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 avoid 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 the 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.

Example: 3-Segment Instruction Pipeline

- A typical RISC processor will have 3 instruction formats:
  - Data manipulation instructions use registers only
  - Data transfer instructions use an effective address consisting of register contents and constant offset
  - Program control instructions use registers and a constant to evaluate the branch address.

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.
- The instruction cycle can consist of 3 suboperations:
  - I – Instruction fetch
  - A – ALU operation (decode the instruction and calculate the result, or the effective operand address or the branch address
  - E – Execute instruction – direct the result to a register, memory or the PC

Pipeline Timing With Data Conflict

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

Data conflict we're using R2 before it is finished loading

Pipeline Timing With Delayed Loading

The compiler inserting a NOP eliminates the data conflict without needed extra hardware support

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

Delayed Branch

The branch instruction here

makes these Nops necessary

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			
No-operation						I	A	E		
No-operation							I	A	E	
Instruction in X								I	A	E

Delayed Branch

Placing the branch instruction here

makes a Nop instruction here unecessary

Clock cycles:	1	2	3	4	5	6	7	8	9	10
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			
Instruction in X						I	A	E		

Vector Processing

- There is a class of computation problems that require far greater computation power than many computers can provide. They can take days or weeks to solve on conventional computers.
- These problems can be formulated in terms of vectors and matrices.
- Computers that can process vectors as a unit can solve these problems much more easily.

Applications

- Computers with vector processing capabilities are useful in problems such as:
  - Long range weather forecasting
  - Petroleum exploration
  - Seismic data analysis
  - Medical diagnosis
  - Aerodynamics and space flight simulation
  - Artificial intelligence and expert systems
  - Mapping the human genome
  - Image processing

Vector Operations

- 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.
- On sequential computers, vector operations are broken down into single computations on subscripted variables. The operations on the entire vector is done by iteration.

Example: Vector Addition

- Adding corresponding elements in two vectors would be performed by the following FORTRAN loop:

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

- This would implemented by the following machine language 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
```

Vector Addition With a Vector Processor

- A computer with vector processing could handle this with one instruction:

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

- This could be performed using a pipelined floating-point adder similar to the one shown earlier.

The instruction format

Operation code	Base address Source 1	Base address Source 2	Base address Destination	Vector length

Matrix Multiplication

- Matrix multiplication is an extremely computationally intensive operation.
  - Multiplying 2 n×n matrices requiring n^2 inner products, each of which requires n multiplications = n^3 multiplications
  - An n × m matrix can be thought of as n row vectors or m column vectors.
- Consider multiplying two 3×3 matrices:

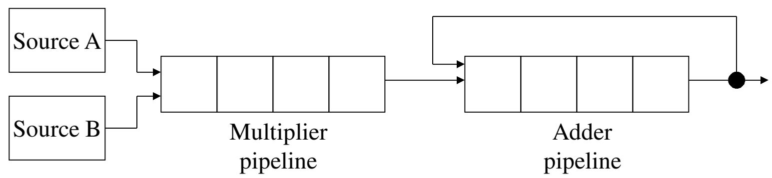
Matrix Multiplication (continued)

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

where $c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$ $\therefore$ c11 = $a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$

This requires 27 multiplication.
It is typical to encounter matrices that have 100 or even 1000 rows and/or columns.

Pipeline for Calculating An Inner Product

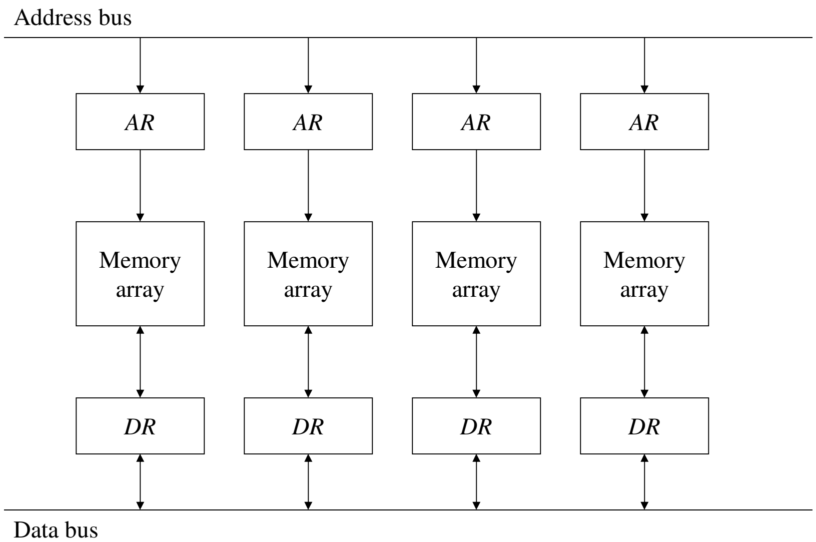


$$
\begin{aligned}
C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \dots \\
& + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \dots \\
& + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \dots \\
& + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \dots
\end{aligned}
$$

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

- Supercomputers are commercial computers with vector instructions and pipeline floating-point arithmetic operations.
  - Components are also placed in close proximity to speed data transfers and require special cooling because of the resultant heat build-up.
- Supercomputers have all the standard instructions that one might expect as well as those for vector operations. They have multiple functional units, each with their own pipeline.
- They also make heavy use of parallel processing and are optimized for large-scale numerical operations.

Supercomputers and Performance

- Supercomputer performance are usually measured in terms of floating-point operations per second (***flops***). Derivative terms include ***megaflops*** and ***gigaflops***.
- Typical supercomputer has a cycle time of 4 to 20 ns.
  - With one floating-point operation per cycle, this can lead to performance of 50 to 250 megaflops. (This does not include set-up time).

Cray 1

- The Cray 1 was the first supercomputer (1976).

  It used vector processing with 12 distinct functional units in parallel, operating concurrently with operands stored in over 150 registers.

  It could perform a floating-point operation on 2 sets of 64 operands in one 12.5 ns clock cycle, translating to 80 megaflops.

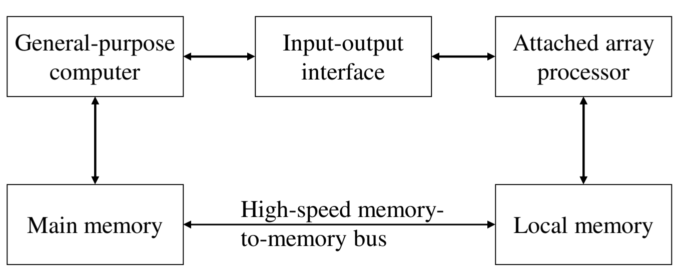
Array Processors

- 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 Processors

- An attached array processor is designed as a peripheral for a conventional host computer, providing vector processing for complex numerical applications.
- The array processor, working with multiple functional units, serves as a back-end machine driven by the host computer.

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's main purpose is to decode instruction and determine how they are executed.

SIMD Array Processor Organization