Forms of Computing

Monolithic computing

Monolithic computing is the simplest form of computing. A single computer such as personal computer is used for computing. The computer that does monolithic computing is not connected to any network.

There are two types: single user monolithic computing and multi-user monolithic computing.

Single user monolithic computing

In this type of computing, the computer is used by only one user at a time. The computer can use resources only within its immediate access. Applications such as spreadsheet or word processing program practise this form of computing.

Multi-user monolithic computing

In this type of computing, multiple users can share the resources using a technique called timesharing. Usually mainframe is used to prove the centralized resource. Users can access mainframe system through terminals. Applications such as payroll and billing practise this form of computing.

Distributed Computing

Distributed Computing is performed among many computers connected via network. Each computer has one more processor and other resources. User's workstation is connected to local computer so that the user can fully use resources on that computer. Moreover the user can access resources on the remote computers via network. Example for this type is World Wide Web.

Parallel computing

Parallel computing or parallel processing uses multiple processors at the same time to execute a single program. This type of computing can exhibit its full potential only when the program can be split into many pieces so that each CPU can execute a portion. This is typically performed on a single computer having multiple CPUs. Parallel processing is useful for solving computing-intensive problems. So it is used in areas such as biology, airspace and semiconductor design.

Parallel Computer Architecture

Classification

PIPELINE PROCESSING

Instruction Pipelines

As discussed earlier, the stream of instructions in the instruction execution cycle can be realized through a pipeline where overlapped execution of different operations are performed. The process of executing the instruction involves the following major steps:

- Fetch the instruction from the main memory
- Decode the instruction
- Fetch the operand
- Execute the decoded instruction

These four steps become the candidates for stages for the pipeline, which we call as instruction pipeline (It is shown in *Figure 3*).

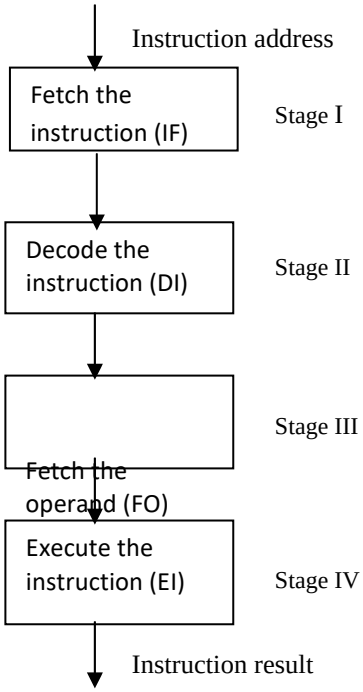


Figure 3: Instruction Pipeline

Since, in the pipelined execution, there is overlapped execution of operations, the four stages of the instruction pipeline will work in the overlapped manner. First, the instruction address is fetched from the memory to the first stage of the pipeline. The first stage fetches the instruction and gives its output to the second stage. While the second stage of the pipeline is decoding the instruction, the first stage gets another input and fetches the next instruction. When the first instruction has been decoded in the second stage, then its output is fed to the third stage. When the third stage is fetching the operand for the first instruction, then the second stage gets the second instruction and the first stage gets input for another instruction and so on. In this way, the pipeline is executing the instruction in an overlapped manner increasing the throughput and speed of execution.

The scenario of these overlapped operations in the instruction pipeline can be illustrated through the space-time diagram. In *Figure 4*, first we show the space-time diagram for non-overlapped execution in a sequential environment and then for the overlapped pipelined environment. It is clear from the two diagrams that in non-overlapped execution, results are achieved only after 4 cycles while in overlapped pipelined execution, after 4 cycles, we are getting output after each cycle. Soon in the instruction pipeline, the instruction cycle has been reduced to ¼ of the sequential execution.

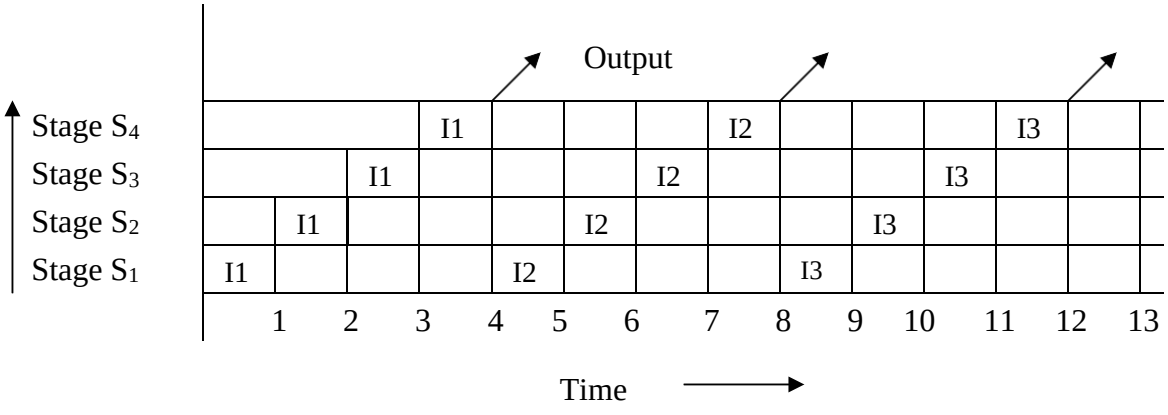


Figure 4(a) Space-time diagram for Non-pipelined Processor

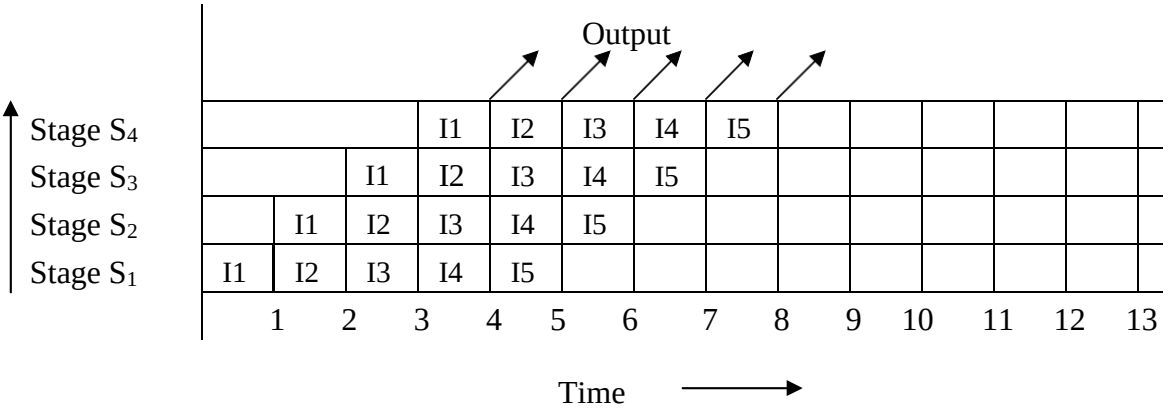


Figure 4(b) Space-time diagram for Overlapped Instruction pipelined Processor

Instruction buffers: For taking the full advantage of pipelining, pipelines should be filled continuously. Therefore, instruction fetch rate should be matched with the pipeline consumption rate. To do this, instruction buffers are used. Instruction buffers in CPU have high speed memory for storing the instructions. The instructions are pre-fetched in the buffer from the main memory. Another alternative for the instruction buffer is the cache memory between the CPU and the main memory. The advantage of cache memory is that it can be used for both instruction and data. But cache requires more complex control logic than the instruction buffer. Some pipelined computers have adopted both.

Arithmetic Pipelines

The technique of pipelining can be applied to various complex and slow arithmetic operations to speed up the processing time. The pipelines used for arithmetic computations are called *Arithmetic pipelines*. In this section, we discuss arithmetic pipelines based on arithmetic operations. Arithmetic pipelines are constructed for simple fixed-point and complex floating-point arithmetic operations. These arithmetic operations are well suited to pipelining as these operations can be efficiently partitioned into subtasks for the pipeline stages. For implementing the arithmetic pipelines we generally use following two types of adder:

i) **Carry propagation adder (CPA)**: It adds two numbers such that carries generated in successive digits are propagated.

ii) **Carry save adder (CSA)**: It adds two numbers such that carries generated are not propagated rather these are saved in a carry vector.

Fixed Arithmetic pipelines: We take the example of multiplication of fixed numbers. Two fixed-point numbers are added by the ALU using add and shift operations. This sequential execution makes the multiplication a slow process. If we look at the multiplication process carefully, then we observe that this is the process of adding the multiple copies of shifted multiplicands as show below:

$X_5 \quad X_4\ X_3\ X_2\ \ X_1\ \ X_0 = X$

$Y_5 \quad Y_4\ Y_3\ Y_2\ Y_1\ Y_0 = Y$

$X_5Y_0\ X_4Y_0\ X_3Y_0\ X_2Y_0\ X_1Y_0\ X_0Y_0 = P_1$

$X_5Y_1\ X_4Y_1\ X_3Y_1\ \ X_2Y_1\ X_1Y_1\ \ X_0Y_1 \qquad = P_2$

$X_5Y_2\ X_4Y_2\ X_3Y_2\ X_2Y_2\ X_1Y_2\ \ X_0Y_2 \qquad = P_3$

$X_5Y_3\ X_4Y_3\ X_3Y_3\ X_2Y_3\ X_1Y_3\ X_0Y_3 \qquad = P_4$

$X_5Y_4\ X_4Y_4\ X_3Y_4\ X_2Y_4\ X_1Y_4\ X_0Y_4 \qquad = P_5$

$X_5Y_5\ X_4Y_5\ X_3Y_5\ X_2Y_5\ X_1Y_5\ X_0Y_5 \qquad = P_6$

Now, we can identify the following stages for the pipeline:

- The first stage generates the partial product of the numbers, which form the six rows of shifted multiplicands.
- In the second stage, the six numbers are given to the two CSAs merging into four numbers.
- In the third stage, there is a single CSA merging the numbers into 3 numbers.
- In the fourth stage, there is a single number merging three numbers into 2 numbers.
- In the fifth stage, the last two numbers are added through a CPA to get the final product.

These stages have been implemented using CSA tree as shown in *Figure 5*.

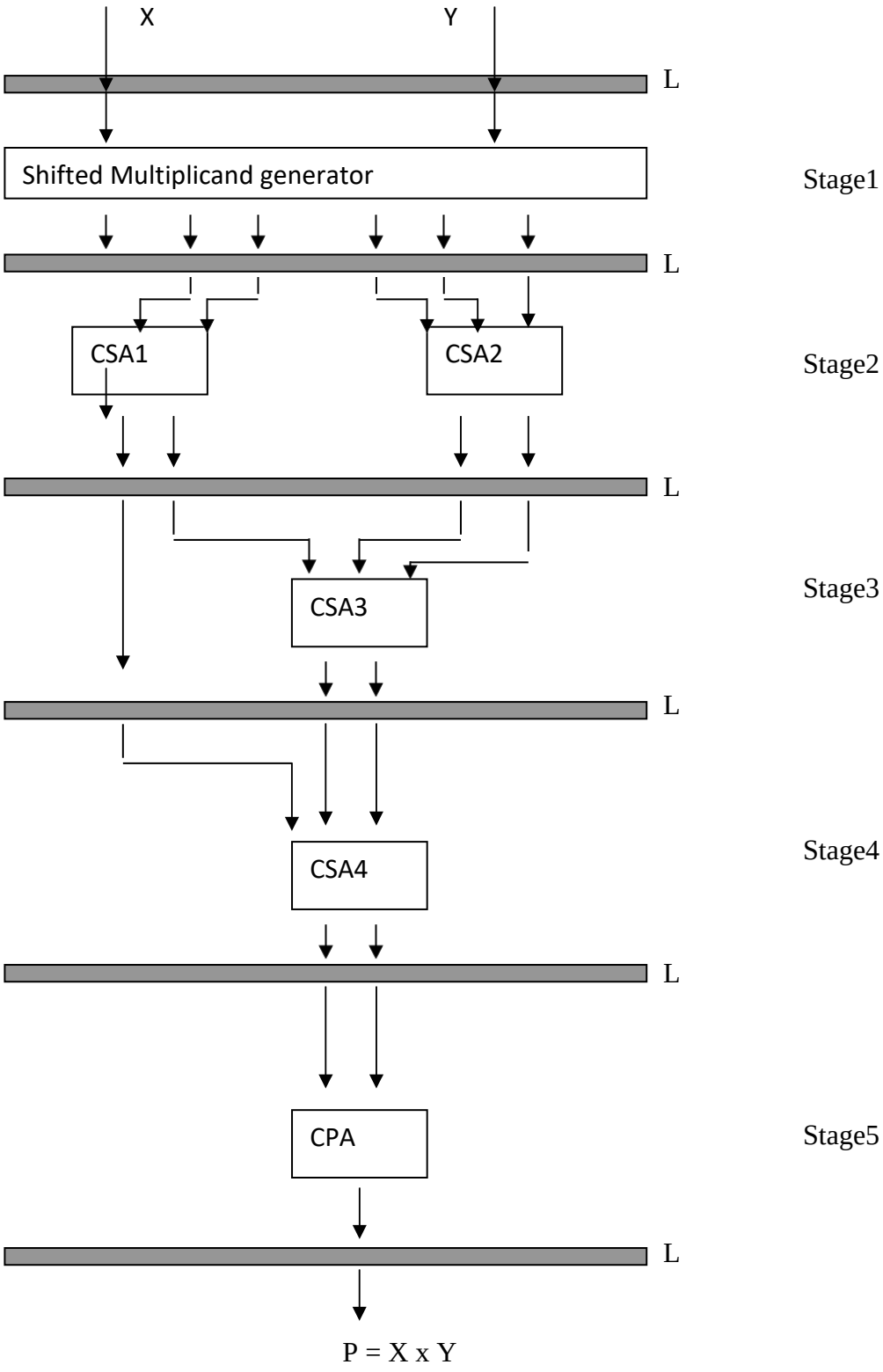


Figure 5: Arithmetic pipeline for Multiplication of two 6-digit fixed numbers

VECTOR PROCESSING

A *vector* is an ordered set of the same type of scalar data items. The scalar item can be a floating pint number, an integer or a logical value. *Vector processing* is the arithmetic or logical computation applied on vectors whereas in scalar processing only one or pair of data is processed. Therefore, vector processing is faster compared to scalar processing. When the scalar code is converted to vector form then it is called *vectorization*. A *vector processor* is a special coprocessor, which is designed to handle the vector computations.

Vector instructions can be classified as below:

- **Vector-Vector Instructions**: In this type, vector operands are fetched from the vector register and stored in another vector register. These instructions are denoted with the following function mappings:

  F1 : V -> V

  F2 : V × V -> V

  For example, vector square root is of F1 type and addition of two vectors is of F2.
- **Vector-Scalar Instructions**: In this type, when the combination of scalar and vector are fetched and stored in vector register. These instructions are denoted with the following function mappings:

  F3 : S × V -> V where S is the scalar item

  For example, vector-scalar addition or divisions are of F3 type.
- **Vector reduction Instructions**: When operations on vector are being reduced to scalar items as the result, then these are vector reduction instructions. These instructions are denoted with the following function mappings:

  F4 : V -> S

  F5 : V × V -> S

- **Vector-Memory Instructions**: When vector operations with memory M are performed then these are vector-memory instructions. These instructions are denoted with the following function mappings:

  F6 : M-> V
  F7 : V -> V

  For example, vector load is of type F6 and vector store operation is of F7.

Vector Processing with Pipelining: Since in vector processing, vector instructions perform the same computation on different data operands repeatedly, vector processing is most suitable for pipelining. Vector processors with pipelines are designed to handle vectors of varying length n where n is the length of vector. A vector processor performs better if length of vector is larger. But large values of n causes the problem in storage of vectors and there is difficulty in moving the vectors to and from the pipelines.

Pipeline Vector processors adopt the following two architectural configurations for this problem as discussed below:

- ***Memory-to-Memory Architecture***: The pipelines can access vector operands, intermediate and final results directly in the main memory. This requires the higher memory bandwidth. Moreover, the information of the base address, the offset and vector length should be specified for transferring the data streams between the main memory and pipelines. STAR-100 and TI-ASC computers have adopted this architecture for vector instructions.

- ***Register*-to-Register Architecture**: In this organization, operands and results are accessed indirectly from the main memory through the scalar or vector registers. The vectors which are required currently can be stored in the CPU registers. Cray-1 computer adopts this architecture for the vector instructions and its CPY contains 8 vector registers, each register capable of storing a 64 element vector where one element is of 8 bytes.

ARRAY PROCESSING

We have seen that for performing vector operations, the pipelining concept has been used. There is another method for vector operations. If we have an array of n processing elements (PEs) i.e., multiple ALUs for storing multiple operands of the vector, then an n instruction, for example, vector addition, is broadcast to all PEs such that they add all

operands of the vector at the same time. That means all PEs will perform computation in parallel. All PEs are synchronised under one control unit. This organisation of synchronous array of PEs for vector operations is called *Array Processor*. The organisation is same as in SIMD which we studied in unit 2. An array processor can handle one instruction and multiple data streams as we have seen in case of SIMD organisation. Therefore, array processors are also called *SIMD array computers*.

The organisation of an array processor is shown in *Figure 7*. The following components are organised in an array processor:

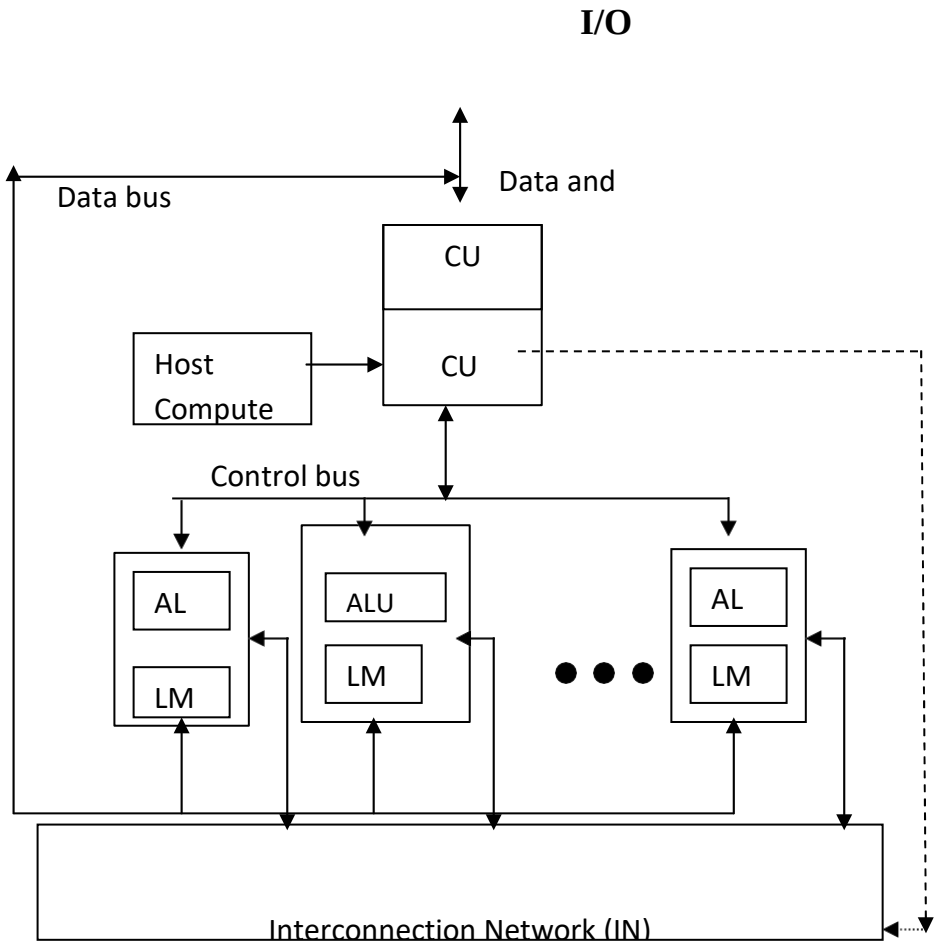


Figure 7: Organisation of SIMD Array Processor

Control Unit (CU) : All PEs are under the control of one control unit. CU controls the inter communication between the PEs. There is a local memory of CU also called CY memory. The user programs are loaded into the CU memory. The vector instructions in the program are decoded by CU and broadcast to the array of PEs. Instruction fetch and decoding is done by the CU only.

Processing elements (PEs) : Each processing element consists of ALU, its registers and a local memory for storage of distributed data. This PEs has been interconnected via an interconnection network. All PEs receive the instructions from the control unit and the different component operands are fetched from their local memory. Thus, all PEs perform the same function synchronously in a lock-step fashion under the control of the CU. It may be possible that all PEs need not participate in the execution of a vector instruction. Therefore, it is required to adopt a masking scheme to control the status of each PE. A *masking vector* is used to control the status of all PEs such that only enabled PEs are allowed to participate in the execution and others are disabled.

Interconnection Network (IN): IN performs data exchange among the PEs, data routing and manipulation functions. This IN is under the control of CU.

Host Computer: An array processor may be attached to a host computer through the control unit. The purpose of the host computer is to broadcast a sequence of vector instructions through CU to the PEs. Thus, the host computer is a general-purpose machine that acts as a manager of the entire system.

Shared-Memory Parallel Computers

Three most common shared memory multiprocessors models are −

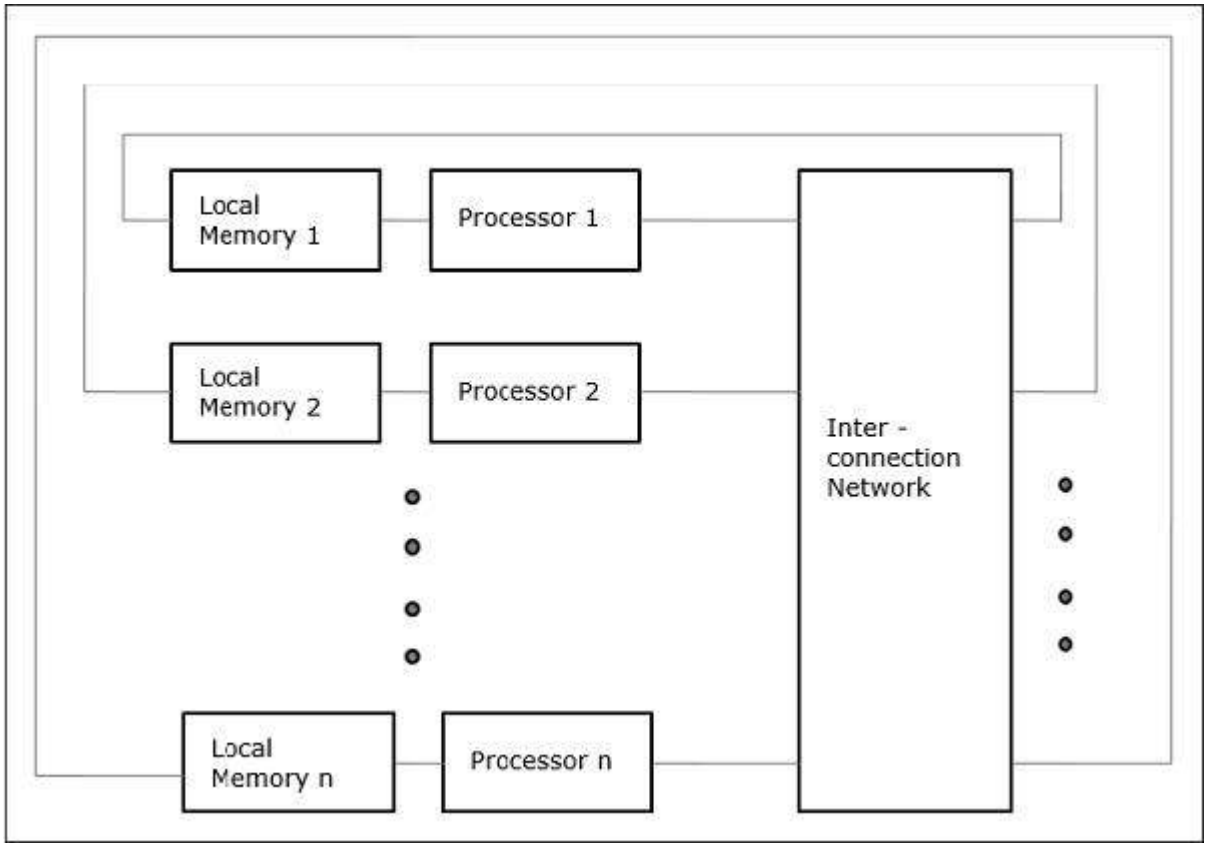
Uniform Memory Access (UMA)

In this model, all the processors share the physical memory uniformly. All the processors have equal access time to all the memory words. Each processor may have a private cache memory. Same rule is followed for peripheral devices.

When all the processors have equal access to all the peripheral devices, the system is called a **symmetric multiprocessor**. When only one or a few processors can access the peripheral devices, the system is called an **asymmetric multiprocessor**.

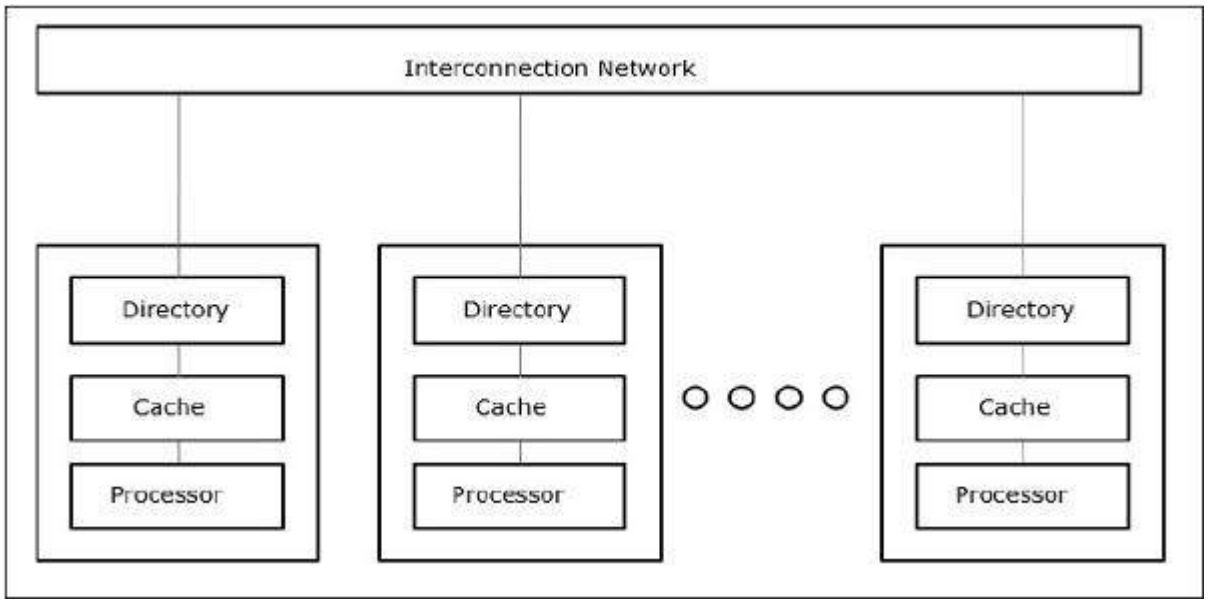
Non-uniform Memory Access (NUMA)

In NUMA multiprocessor model, the access time varies with the location of the memory word. Here, the shared memory is physically distributed among all the processors, called local memories. The collection of all local memories forms a global address space which can be accessed by all the processors.

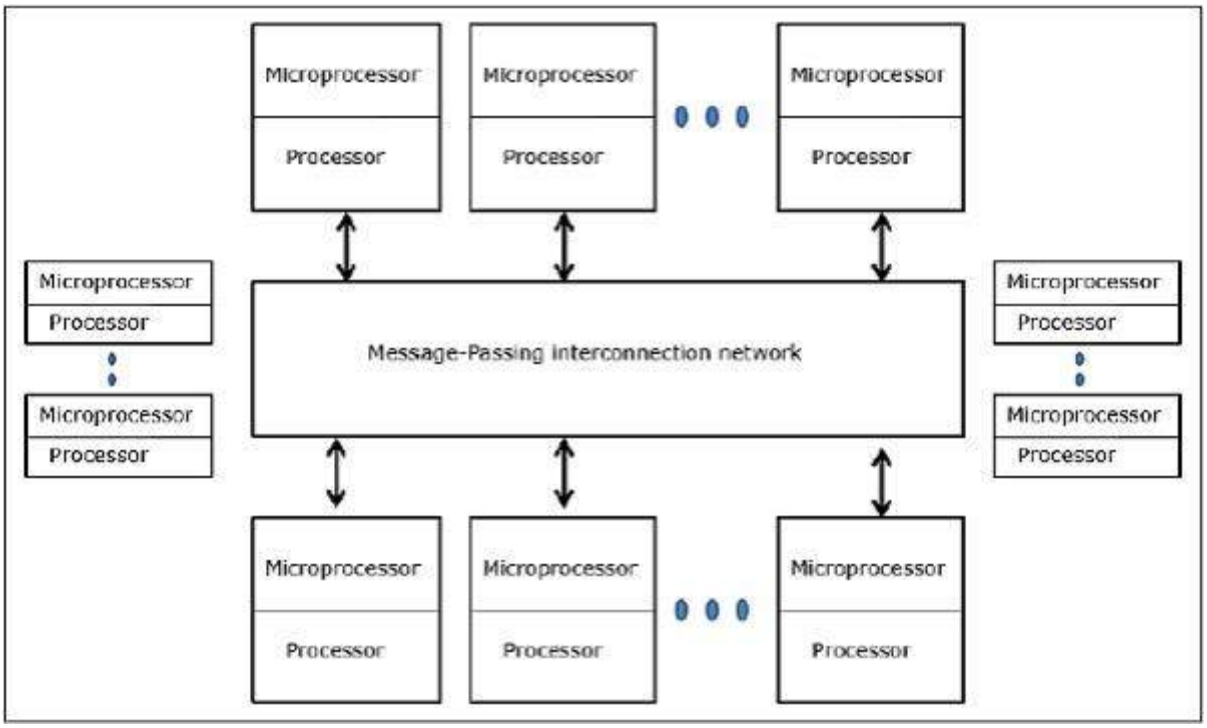


Cache Only Memory Architecture (COMA)

The COMA model is a special case of the NUMA model. Here, all the distributed main memories are converted to cache memories.



- **Distributed - Memory Multicomputer** − A distributed memory multicomputer system consists of multiple computers, known as nodes, inter-connected by message passing network. Each node acts as an autonomous computer having a processor, a local memory and sometimes I/O devices. In this case, all local memories are private and are accessible only to the local processors. This is why; the traditional machines are called **no-remote-memory-access (NORMA)** machines.



Distributed Shared Memory abbreviated as **DSM** is the implementation of shared memory concept in distributed systems. The DSM system implements the shared memory models in loosely coupled systems that are deprived of a local physical shared memory in the system. In this type of system distributed shared memory provides a virtual memory space that is accessible by all the system (also known as **nodes**) of the distributed hierarchy.

Some common challenges that are to be kept in mind while the implementation of DSM −

- Tracking of the memory address (location) of data stored remotely in shared memory.
- To reduce the communication delays and high overhead associated with the references to remote data.
- Controlling the concurrent access of the data shared in DSM.

Based on these challenges there are algorithms designed to implement distributed shared memory. There are four algorithms −

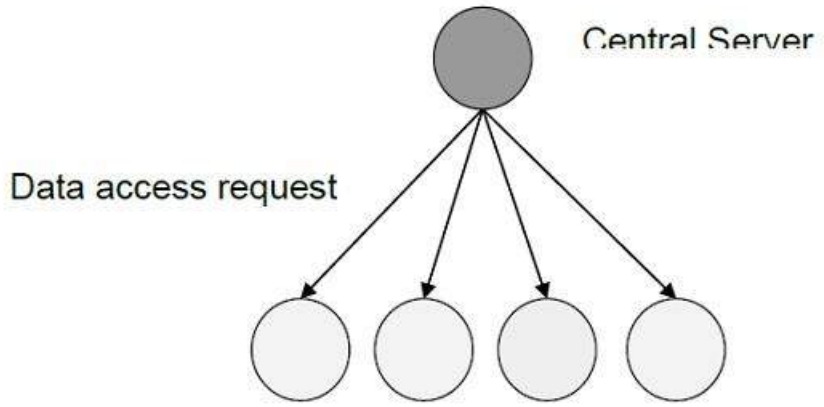
- **Central Server Algorithm**
- **Migration Algorithm**
- **Read Replication Algorithm**
- **Full Replication Algorithm**

Central Server Algorithm

All shared data is **maintained by the central server**. Other nodes of the distributed system **request for reading and writing data** to the server which serves the request and updates or provides access to the data along with **acknowledgment messages**.

These acknowledgment messages are used to provide the status of the data request is served by the server. When the data is sent to the calling function, it acknowledges a number that shows the access sequence of the data to maintain concurrency. And time-out is returned in case of failure.

For larger distributed systems, there can be more than one server. In this case, the servers are located using their address or using mapping functions.

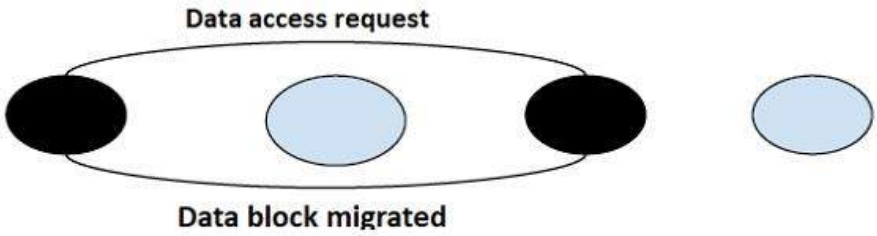


Migration Algorithm

As the name suggest the migration algorithm does the work of migration of data elements. Instead of using a central server serving each request, the **block containing the data requested by a system is migrated** to it for further access and processing. It migrates the data on request.

This algorithm though is good if when a system accesses the same block of data multiple times and the **ability to integrate virtual memory** concept, has some shortcomings that are needed to be addressed.

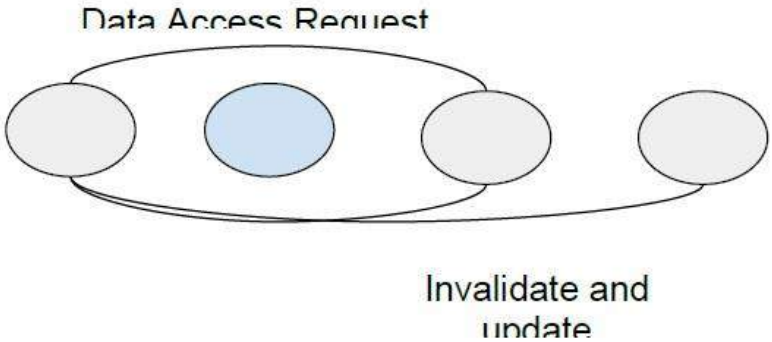
Only one node is able to access the shared data element at a time and the whole block is migrated to that node. Also, this algorithm is more **prone to thrashing** due to the migration of data items upon request by the node.



Read Replication Algorithm

In the read replication algorithm, the data block that is to be accessed is **replicated** and only **reading is allowed** in all the copies. If a write operation is to be done, then all read access is put on halt till all the copies are updated.

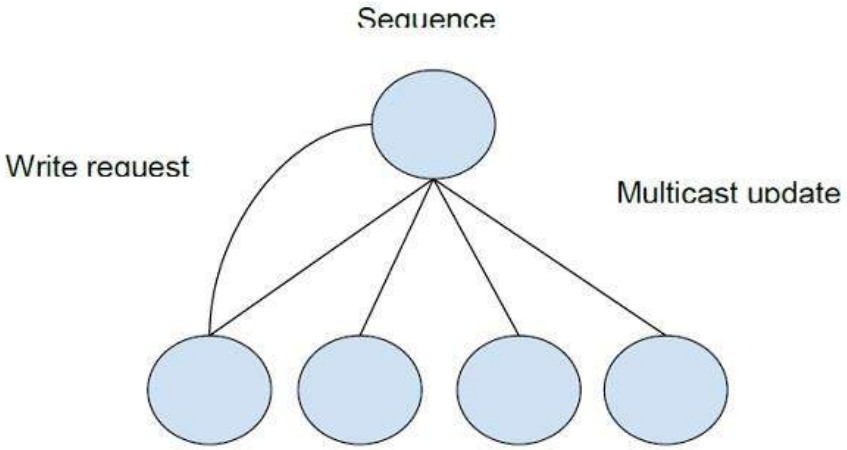
Overall system performance is improved as **concurrent access is allowed**. But write **operation is expensive** due to the requirement of updating all blocks that are shared to maintain concurrency. All copies of data element are to be tracked to maintain consistency.



Full Replication Algorithm

An extension to read the replication algorithm allowing the nodes to perform both **read and writes** operation on the shared block of concurrently. But this access of nodes is controlled to maintain its consistency.

To maintain consistency of data on concurrent access of all nodes sequence is maintained and after every modification that is made in the **data a multicast** with modifications is reflected all the data copies.



- Flynn’s classification of computers
- SIMD Computer System
- MISD Computer System
- MIMD Computer System

Flynn's classification of computers

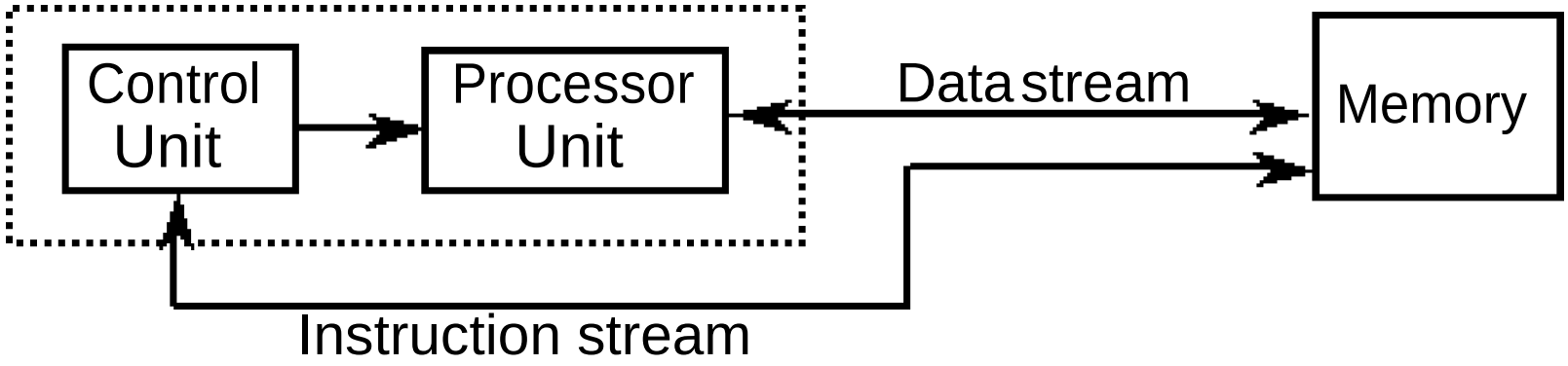
- Michael J Flynn classified computers on the basis of multiplicity of instruction stream and data streams in a computer system.
- It gives how sequence of instructions or data will be executed upon a single processor
- **Instruction stream:** is the sequence of instructions as executed by the machine
- **Data Stream** is a sequence of data including input, or partial or temporary result, called by the instruction Stream.

The four classifications defined by Flynn are based upon the number of concurrent instruction (or control) and data streams available in the architecture:

Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

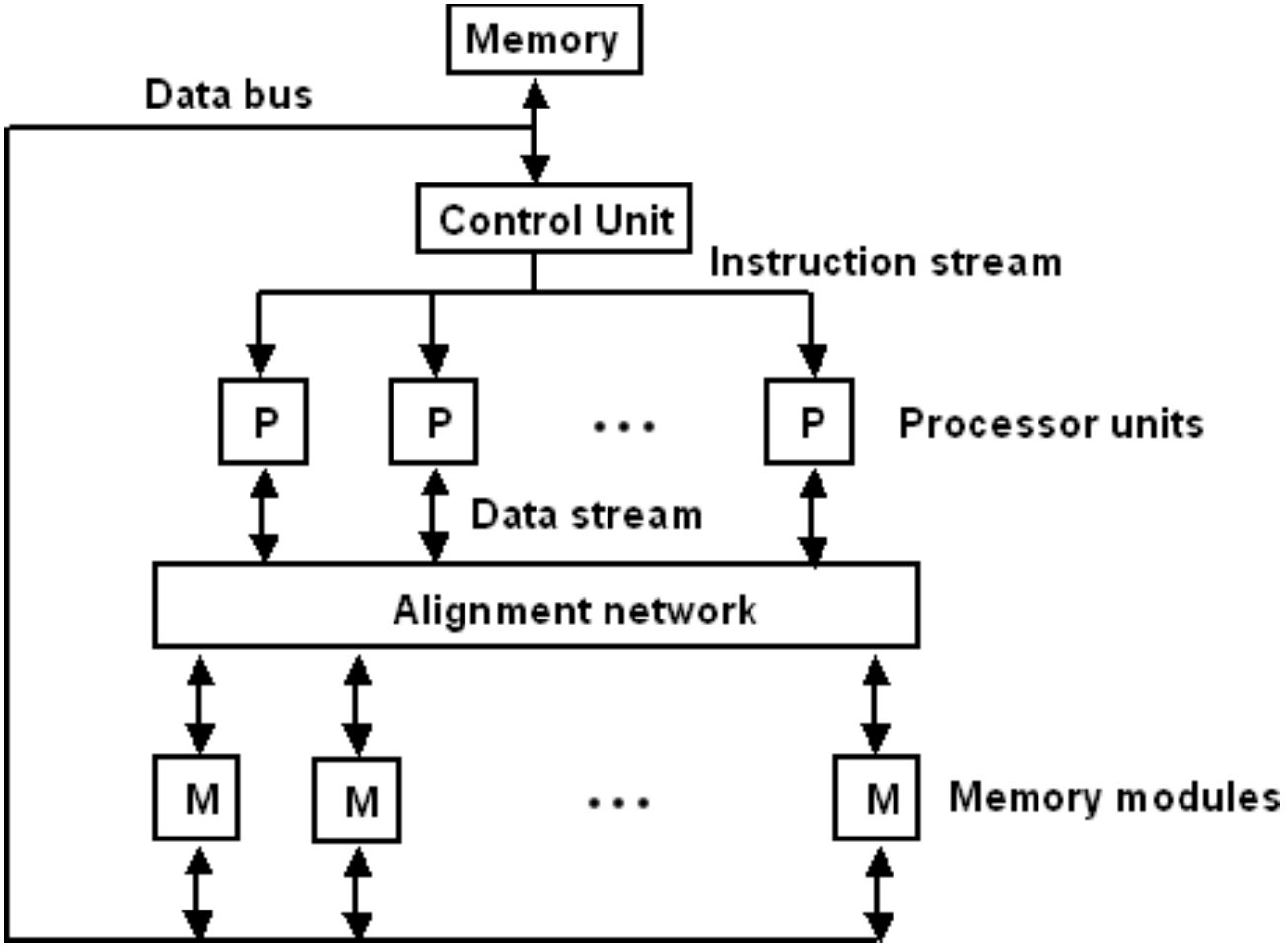
- Instructions are decoded by the control unit and then ctrl unit send the instructions to the processing units for execution.
- Data Stream flows between the processors and memory bi directionally.


Control Unit
Processor Unit
Data stream
Memory
Instruction stream

- The **control unit** directs the various components of a computer. It **reads and interprets (decodes) instructions in the program one by one.**
- A key component of a computer that the control units **interacts with is the program counter**, a special memory (a register) that **keeps track of which location in memory the next instruction is to be read from.**
- Another key component that the control unit **interacts with is the ALU** (arithmetic and logic unit). The ALU is the only part in the main computer that can do any maths. **The control units calls the ALU when it wants to e.g. add two numbers**.
- Finally, the control unit **routinely interacts with the memory**. Program instructions are stored in memory, at the location indicated by the program counter. When the **control unit decodes an instruction, it will often have to fetch some data that also lies in memory.**

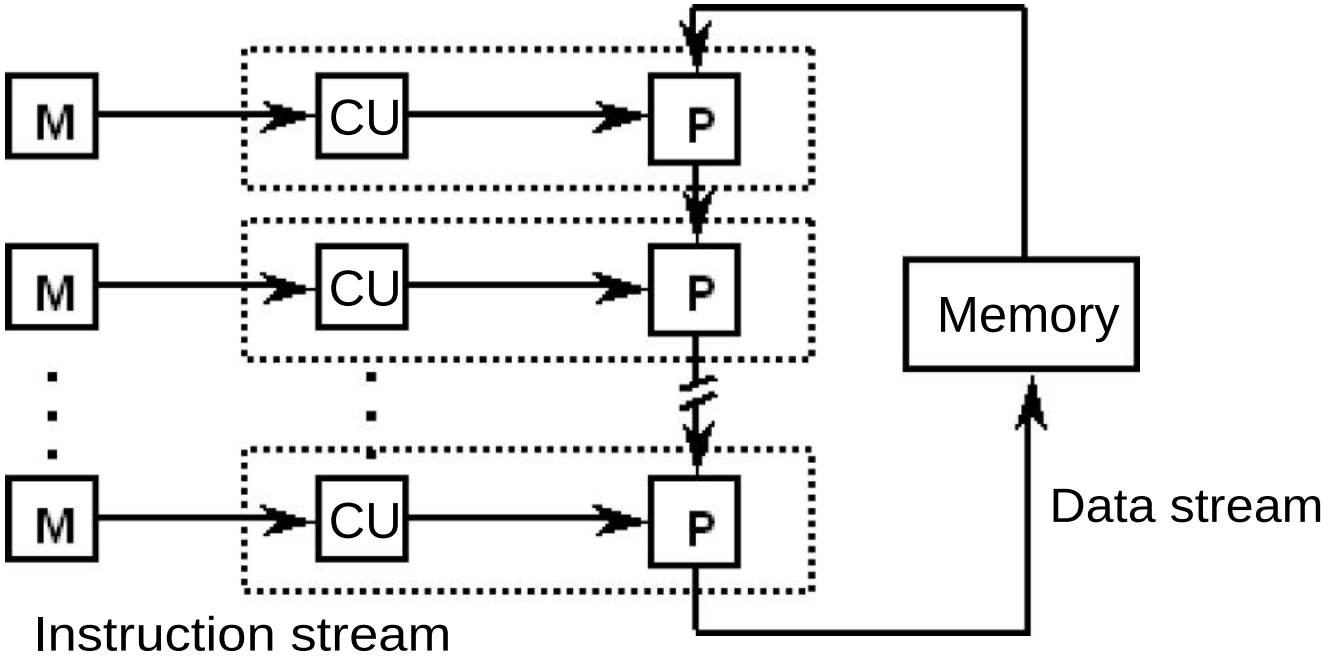
- A sequential computer which exploits no parallelism in either the instruction or data streams.
- Examples of SISD architecture are the traditional uniprocessor machines like a PC or old mainframes.

SIMD Computer System



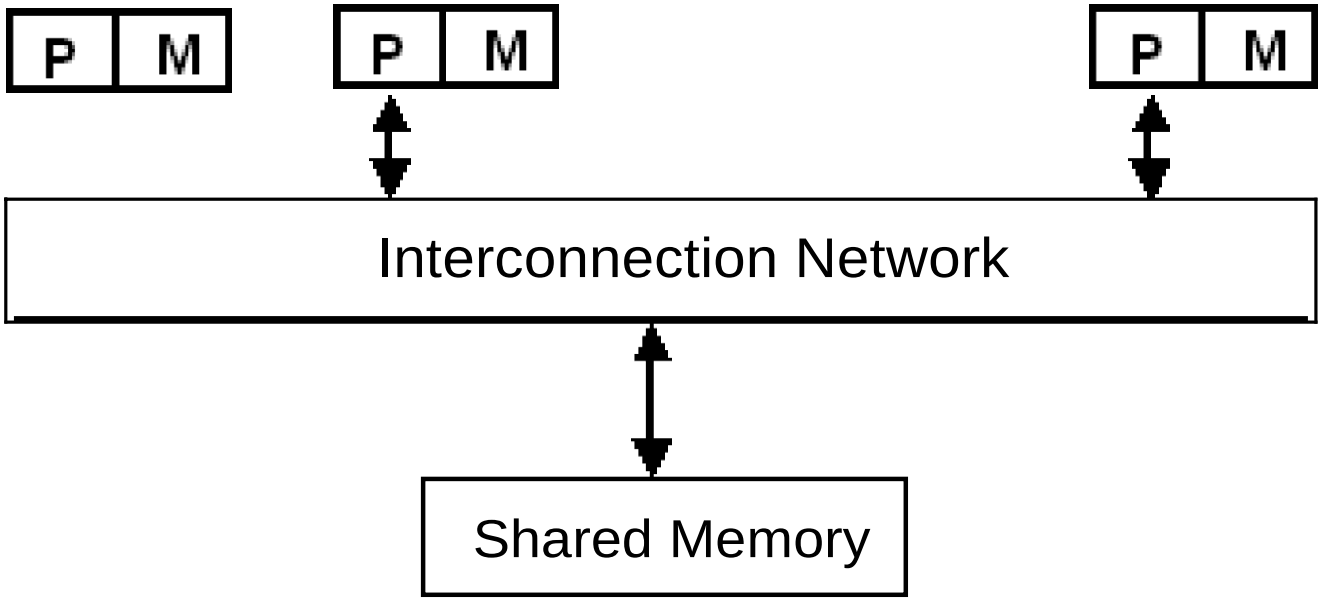
- A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized.
- For example, an array processor.

MISD COMPUTER SYSTEMS



- Multiple instructions operate on a single data stream.
- Uncommon architecture which is generally used for fault tolerance.
- Heterogeneous systems operate on the same data stream and must agree on the result.
- Examples include the Space Shuttle flight control computer.

MMD COMPUTER SYSTEMS



- Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space.

Interconnection Structures

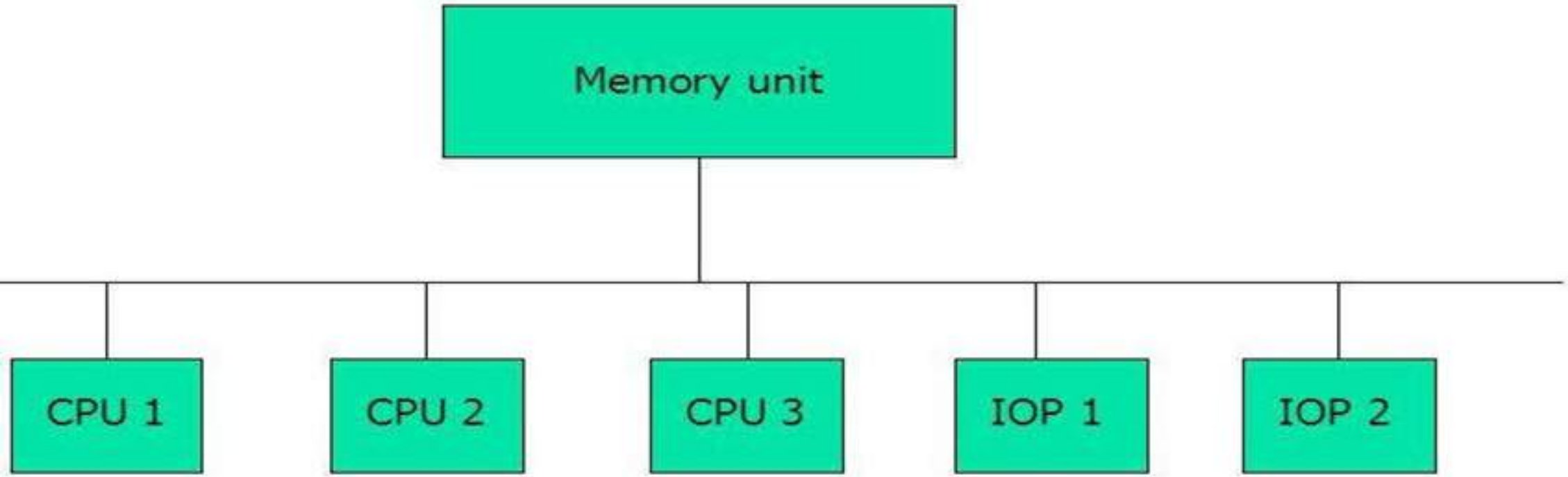
- The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit.
- The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available
  - Between the processors and memory in a shared memory system
  - Among the processing elements in a loosely coupled system

- There are several physical forms available for establishing an interconnection network.
  - Time-shared common bus
  - Multiport memory
  - Crossbar switch
  - Multistage switching network
  - Hypercube system

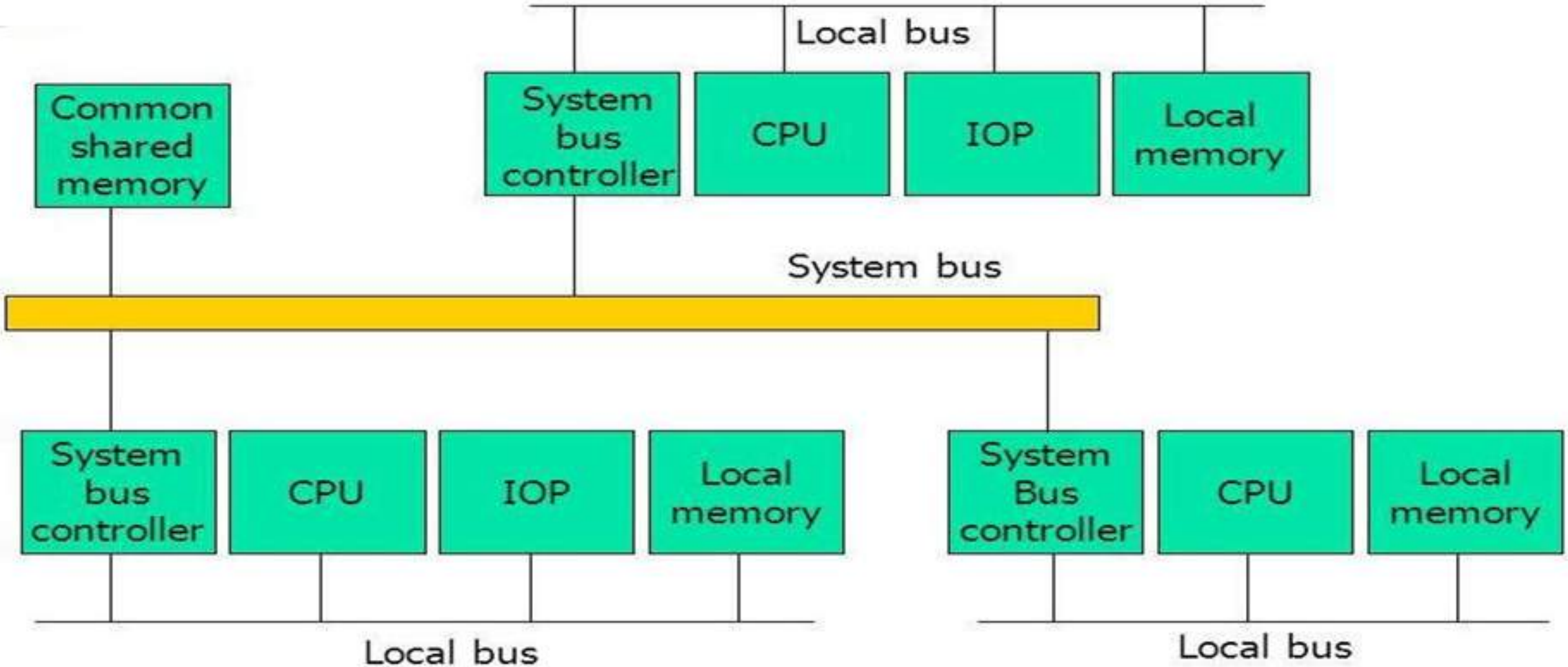
Time-shared common bus

- A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.
- *Disadv.*:
  - Only one processor can communicate with the memory or another processor at any given time.
  - As a consequence, the total overall transfer rate within the system is limited by the speed of the single path
- A more economical implementation of a dual bus structure is depicted in Fig.
- Part of the local memory may be designed as a *cache memory* attached to the CPU.

Time-shared common bus organization



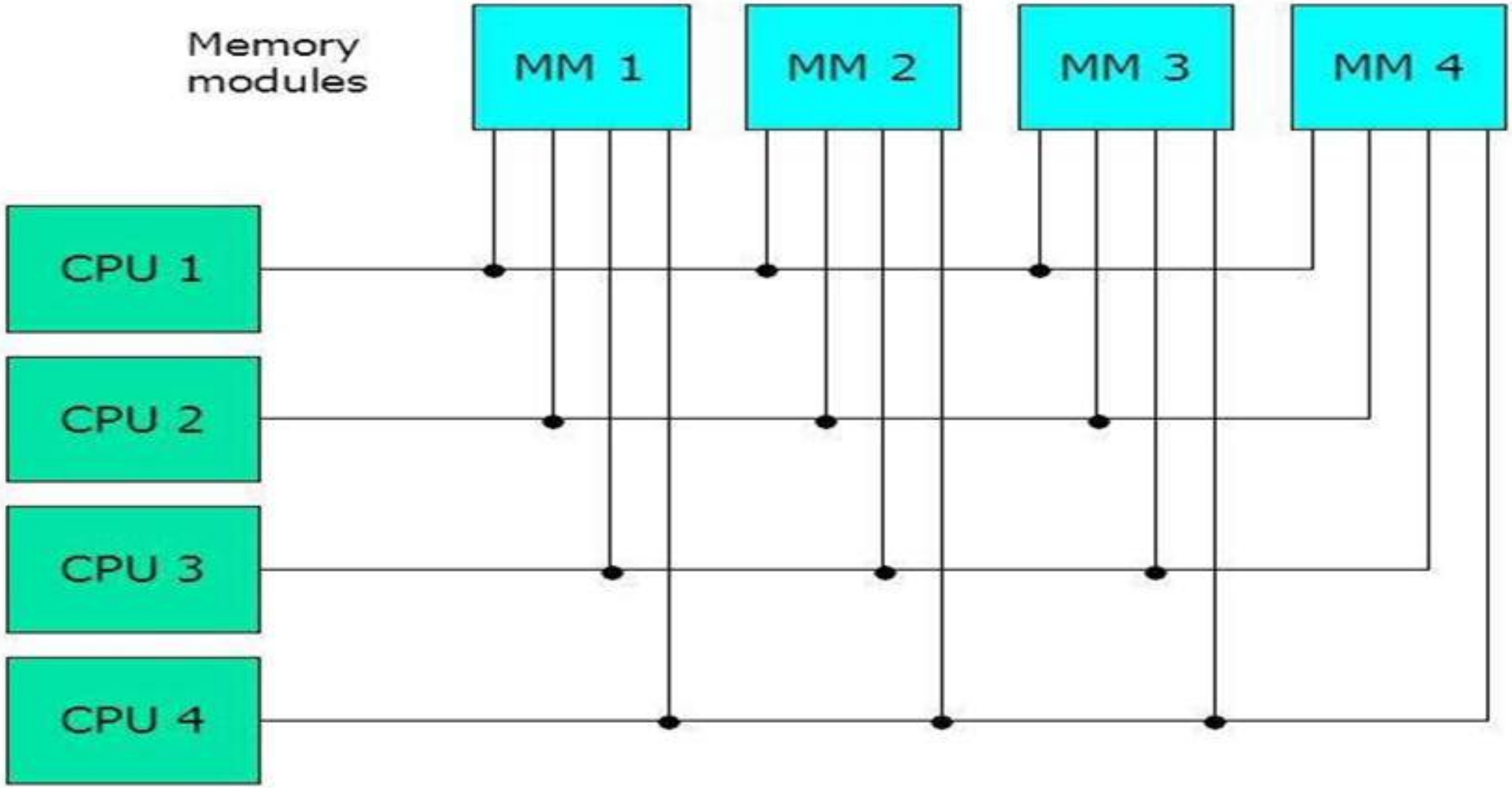
System bus structure for multiprocessors



Multiport memory

- A multiport memory system employs separate buses between each memory module and each CPU.
- The module must have internal control logic to determine which port will have access to memory at any given time.
- Memory access conflicts are resolved by assigning fixed priorities to each memory port.
- *Adv.*:
  - The high transfer rate can be achieved because of the multiple paths.
- *Disadv.*:
  - It requires expensive memory control logic and a large number of cables and connections

Multiport memory organization

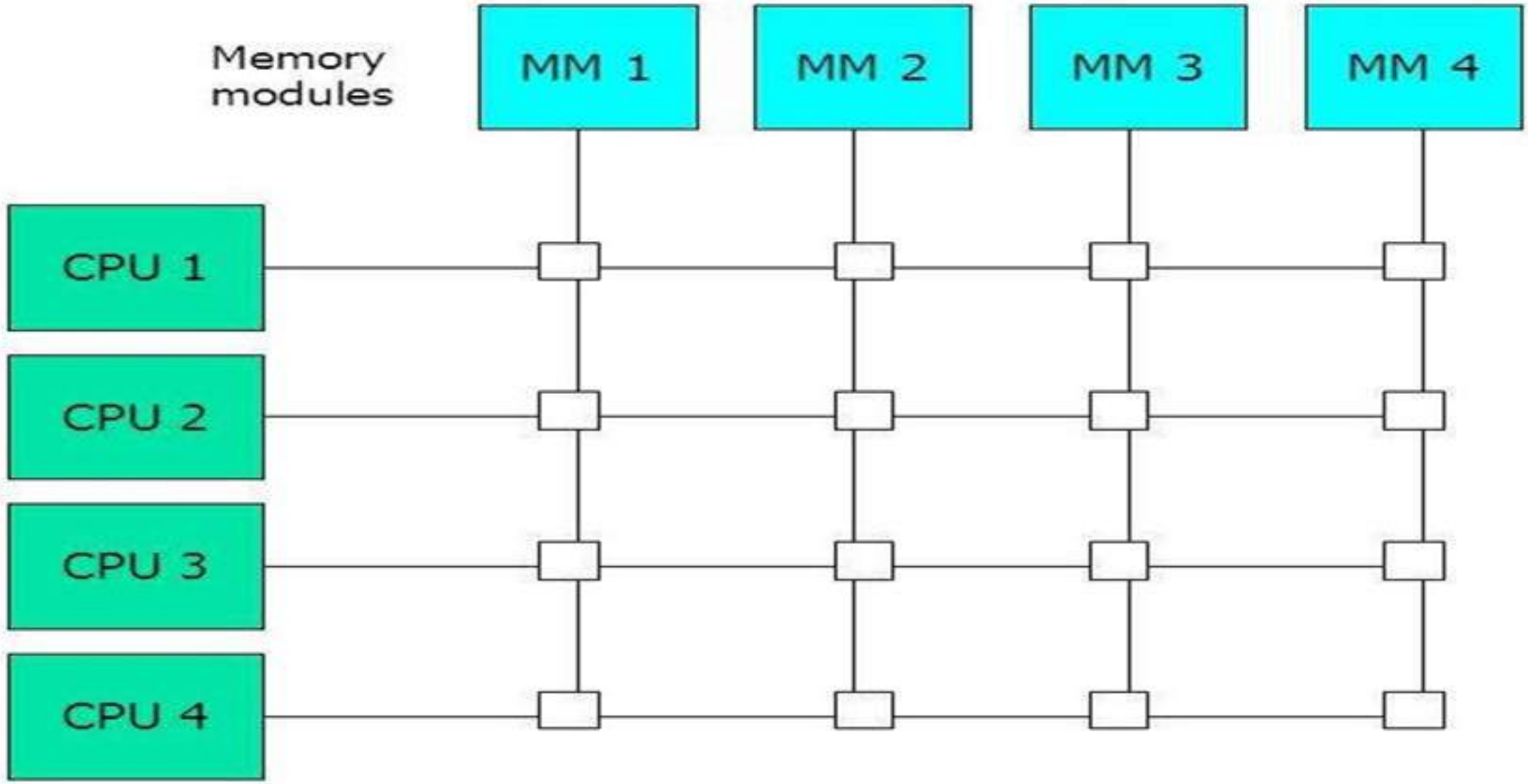


Crossbar switch

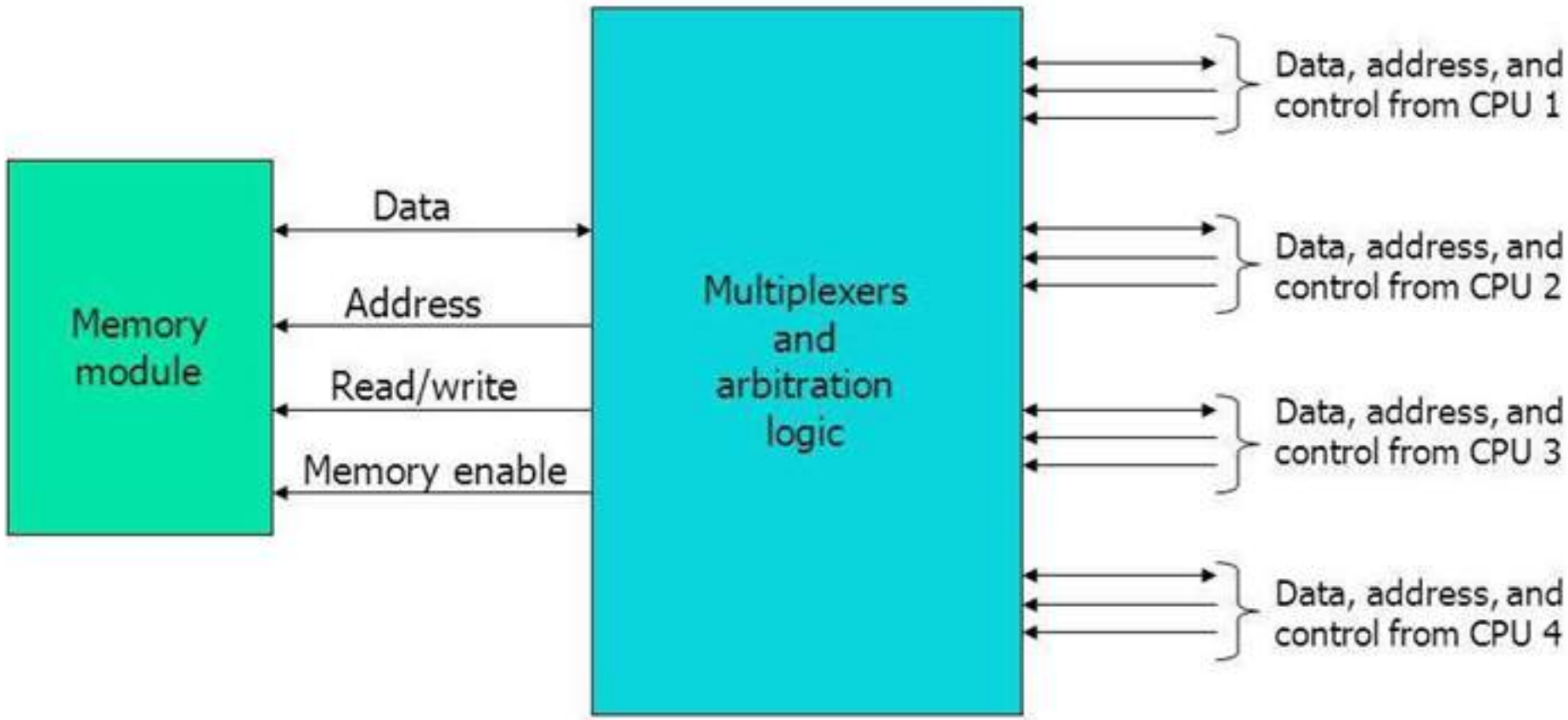
Consists of a number of *crosspoints* that are placed at intersections between processor buses and memory module paths.

- The small square in each crosspoint is a *switch* that determines the path from a processor to a memory module.
- Adv.:
  - Supports simultaneous transfers from all memory modules
- Disadv.:
  - The hardware required to implement the switch can become quite large and complex.
- Fig. shows the functional design of a crossbar switch connected to one memory module.

Crossbar switch



Crossbar switch



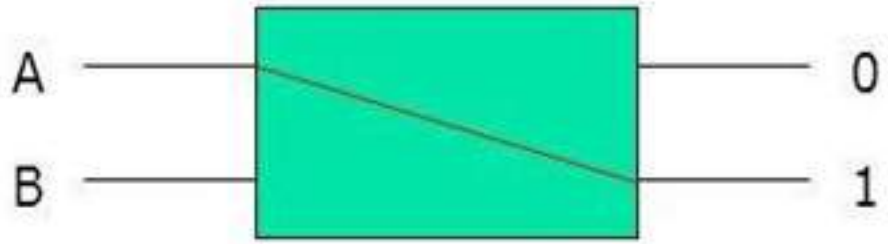
Multistage switching network

- The basic component of a multistage network is a two-input, two-output interchange switch as shown in Fig. .
- Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations.
  - To see how this is done, consider the binary tree shown in Fig
  - Certain request patterns cannot be satisfied simultaneously. i.e., if $P_1 \rightarrow 000 \sim 011$, then $P_2 \rightarrow 100 \sim 111$

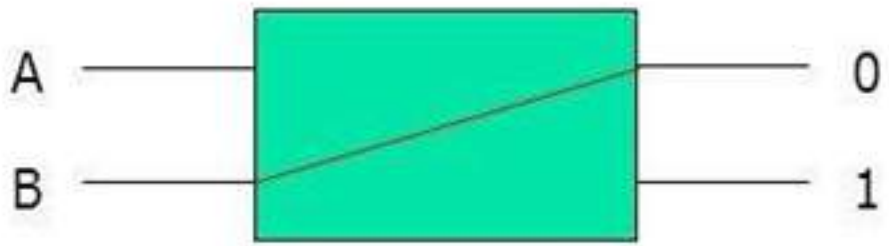
- One such topology is the omega switching net[illegible] shown in Fig.
  - Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001
- In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module.
  - Set up the path → transfer the address into memory → transfer the data
- In a loosely coupled multiprocessor system, both the source and destination are processing elements.

Operation of a 2x2 interchange switch

A connected to 0



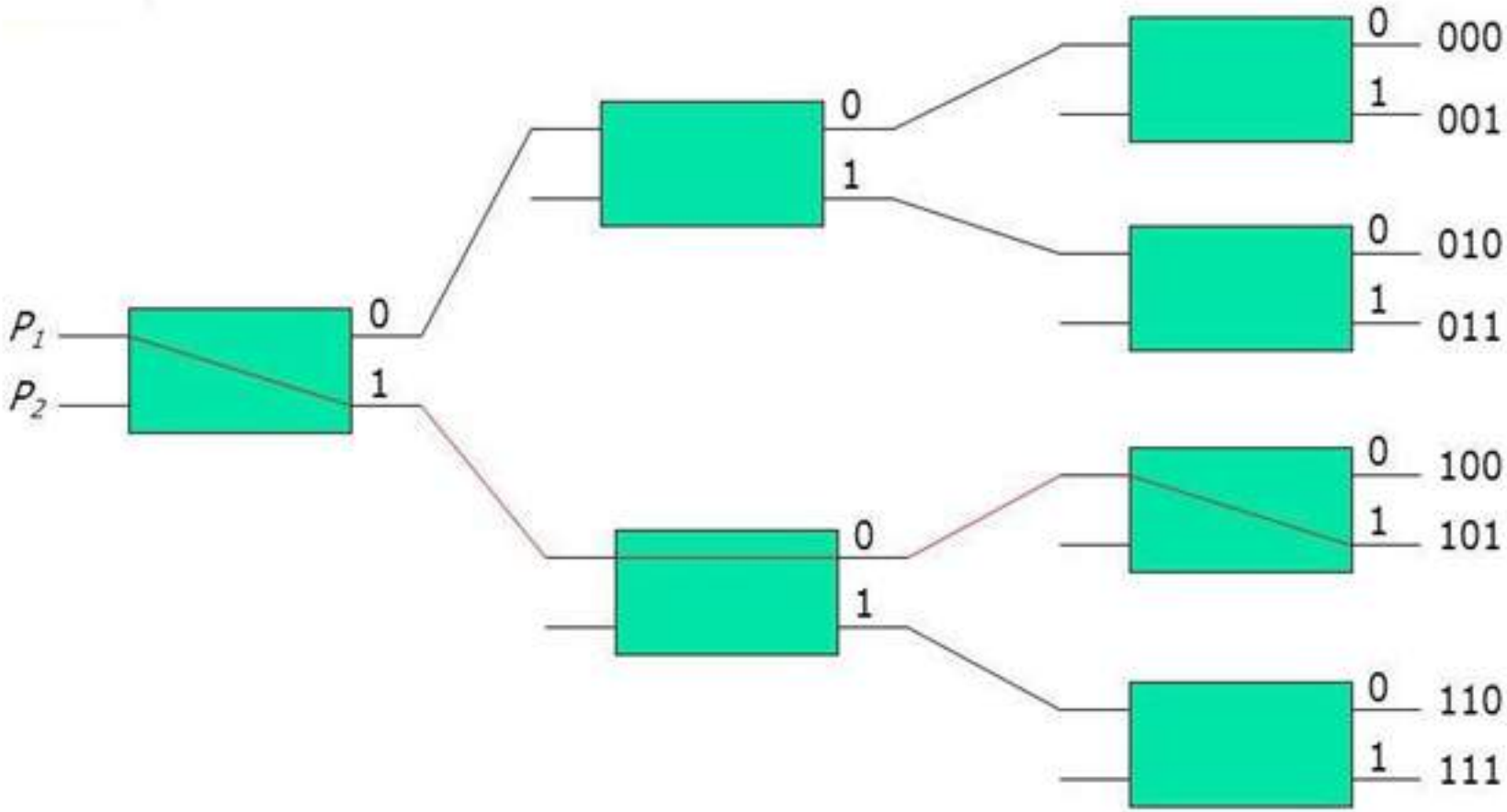
A connected to 1



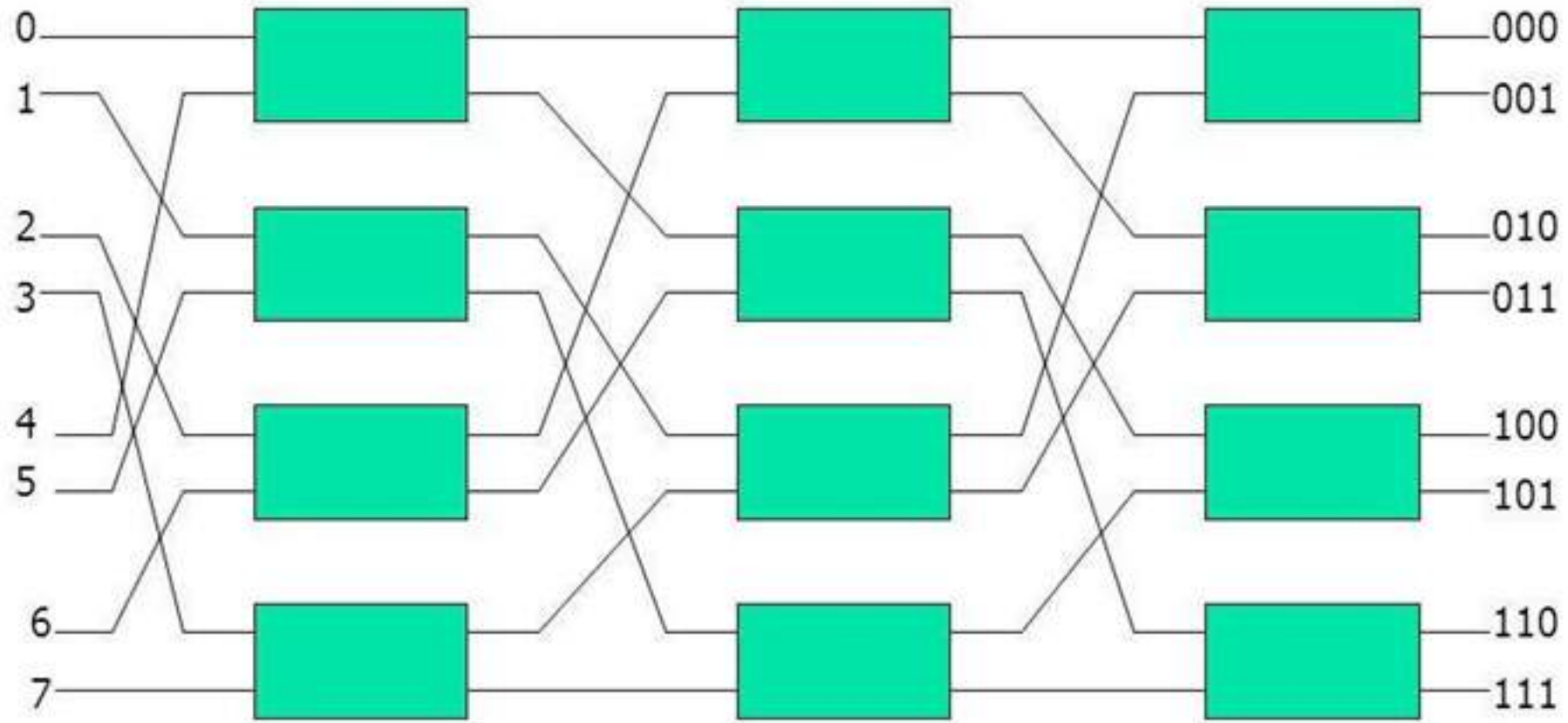
B connected to 0

B connected to 1

Binary tree with 2x2 switches



8x8 omega switching networking



Hypercube system

- The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N=2^n$ processors interconnected in an n-dimensional binary cube.
  - Each processor forms a node of the cube, in effect it contains not only a CPU but also local memory and I/O interface.
  - Each processor address differs from that of each of its n neighbors by exactly one bit position.
- Fig shows the hypercube structure for n=1, 2, and 3.

- Routing messages through an *n*-cube structure may take from one to *n* links from a source node to a destination node.
  - A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
  - The message is then sent along any one of the axes that the resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ.
- A representative of the hypercube architecture is the Intel iPSC computer complex.
  - It consists of 128(n=7) microcomputers, each node consists of a CPU, a floating-point processor, local memory, and serial communication interface units.

Hypercube structures for n=1, 2, 3