RGPVNOTES.IN

Subject Name: **Advanced Computer Architecture**

Subject Code: **CS-6001**

Semester: **6th**

Department of Computer Science and Engineering

Subject Notes

Subject Code: CS-6001 **Subject Name: Advanced Computer Architecture**

UNIT-1

Flynn's Classification

Flynn's classification distinguishes multi-processor computer architectures according to two independent dimensions of Instruction stream and Data stream. An instruction stream is sequence of instructions executed by machine. And a data stream is a sequence of data including input, partial or temporary results used by instruction stream. Each of these dimensions can have only one of two possible states: Single or Multiple. Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections. Following are the four category of Flynn classification and characteristic feature of each of them.

a) Single Instruction Stream, Single Data Stream (SISD)

The figure 1.1 is represents an organization of simple SISD computer having one control unit, one processor unit and single memory unit.

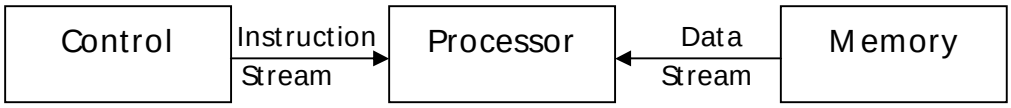


Figure 1.1: SISD processor organizations

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Single data: only one data stream is being used as input during any one clock cycle.
- Deterministic execution.
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer.
- Examples: most PCs, single CPU workstations and mainframes.

b) Single Instruction Stream, Multiple Data Stream (SIMD) processors

- A type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle as shown in figure where there is multiple processors executing instruction given by one control unit.
- Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit.
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Thus single instruction is executed by different processing unit on different set of data as shown in figure 1.2.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.

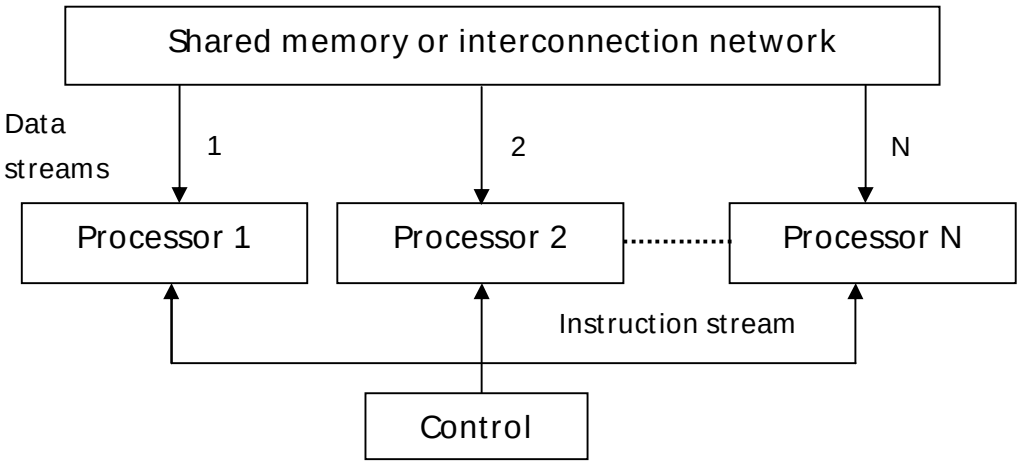


Figure 1.2: SIMD processor organizations

C) Multiple Instruction Stream, Single Data Stream (MISD)

- A single data stream is feed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams as shown in figure 1.3 a single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

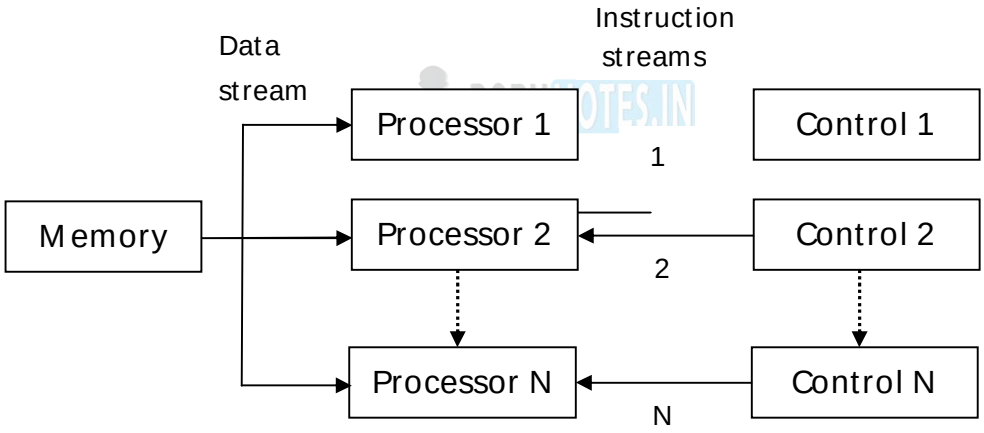


Figure 1.3: MISD processor organizations

- Thus in these computers same data flow through a linear array of processors executing different instruction streams.
- This architecture is also known as systolic arrays for pipelined execution of specific instructions.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).

d) Multiple Instruction Stream, Multiple Data Stream (MIMD)

- Multiple Instructions: Every Processor may be executing a different instruction stream.
- Multiple Data: every processor may be working with a different data stream as shown in figure 1.4 multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor

SMP computers - including some types of PCs.

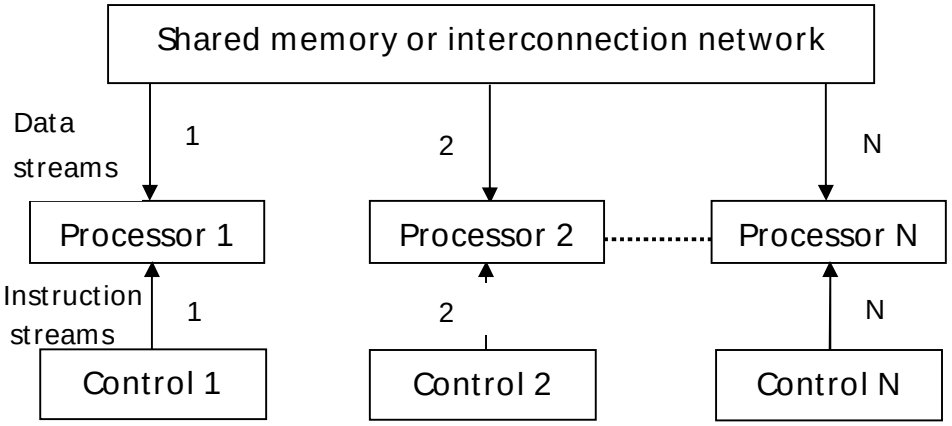


Figure 1.4: MIMD processor organizations

System Attributes to Performance

Performance of a system depends on

- Hardware technology
- Architectural features
- Efficient resource management
- Algorithm design
- Data structures
- Language efficiency
- Programmer skill
- Compiler technology

When we talk about performance of computer system we would describe how quickly a given system can execute a program or programs. Thus we are interested in knowing the turnaround time. Turnaround time depends on:

- Disk and memory accesses
- Input and output
- Compilation time
- Operating system overhead
- CPU time

An ideal performance of a computer system means a perfect match between the machine capability and program behavior. The machine capability can be improved by using better hardware technology and efficient resource management. But as far as program behavior is concerned it depends on code used, compiler used and other run time conditions. Also a machine performance may vary from program to program. Because there are too many programs and it is impractical to test a CPU's speed on all of them benchmarks were developed. Computer architects have come up with a variety of metrics to describe the computer performance.

Clock rate and CPI / IPC: Since I/O and system overhead frequently overlaps processing by other programs, it is fair to consider only the CPU time used by a program, and the user CPU time is the most important factor. CPU is driven by a clock with a constant cycle time (usually measured in nanoseconds, which controls the rate of internal operations in the CPU. The clock mostly has the constant cycle time (t in nanoseconds). The inverse of the cycle time is the clock rate ($f = 1/\tau$, measured in megahertz). A shorter clock cycle time, or equivalently a larger number of cycles per second, implies more operations can be performed per unit time. The size of the program is determined by the instruction count (Ic). The size of a program is determined by its instruction count, Ic, the number of machine instructions to be executed by the program. Different machine instructions require different numbers of clock cycles to execute. CPI (cycles per instruction) is thus

an important parameter.

Average CPI

It is easy to determine the average number of cycles per instruction for a particular processor if we know the frequency of occurrence of each instruction type. Any estimate is valid only for a specific set of programs (which defines the instruction mix), and then only if there are sufficiently large number of instructions.

In general, the term CPI is used with respect to a particular instruction set and a given program mix. The time required to execute a program containing Ic instructions is just $T = Ic * CPI * \tau$.

Each instruction must be fetched from memory, decoded, then operands fetched from memory, the instruction executed, and the results stored.

The time required to access memory is called the memory cycle time, which is usually k times the processor cycle time τ. The value of k depends on the memory technology and the processor-memory interconnection scheme. The processor cycles required for each instruction (CPI) can be attributed to cycles needed for instruction decode and execution (p), and cycles needed for memory references ($m* k$).

The total time needed to execute a program can then be rewritten as

$T = Ic* (p + m*k)*\tau$

Parallel Computer Models

Multiprocessor and Multicomputer

Two categories of parallel computers are discussed below namely shared common memory or unshared distributed memory.

Shared Memory Multiprocessor

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: **UMA, NUMA and COMA**.

Uniform Memory Access (UMA):

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines.
- Identical processors.
- Equal access and access times to memory.
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

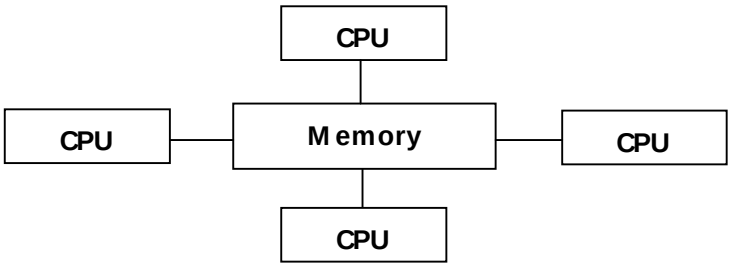


Figure 1.5: Shared Memory (UMA)

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower

If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

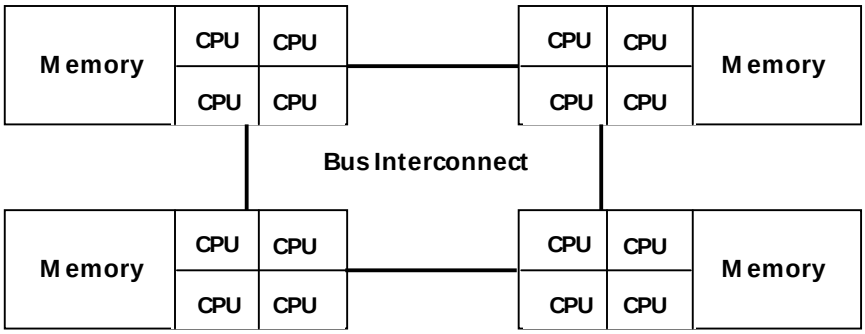


Figure 1.6: Shared Memory (NUMA)

The COMA model (Cache only Memory Access): The COMA model is a special case of NUMA machine in which the distributed main memories are converted to caches. All caches form a global address space and there is no memory hierarchy at each processor node.

Advantages:

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.

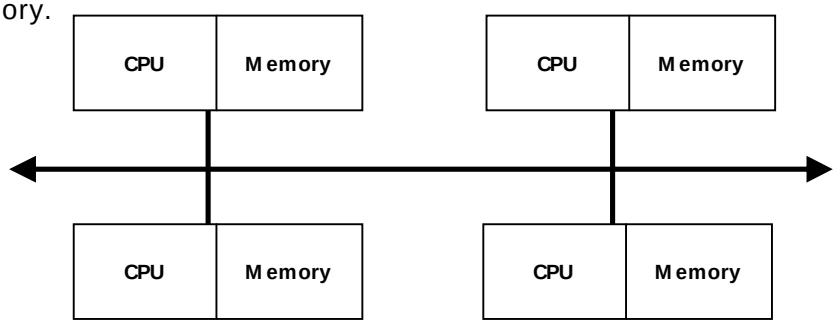


Figure 1.7: Distributed Memory Systems

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently.
- Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- Modern multicomputer use hardware routers to pass message.

Advantages:

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.

Multi-vector and SIMD Computers

A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be a floating point number, an integer, a logical value or a character.

A vector processor consists of a scalar processor and a vector unit, which could be thought of as an independent functional unit capable of efficient vector operations.

Vector Supercomputer

Vector computers have hardware to perform the vector operations efficiently. Operands cannot be used directly from memory but rather are loaded into registers and are put back in registers after the operation. Vector hardware has the special ability to overlap or pipeline operand processing. Vector functional units pipelined, fully segmented each stage of the pipeline performs a step of the function on different operand(s) once pipeline is full; a new result is produced each clock period (cp).

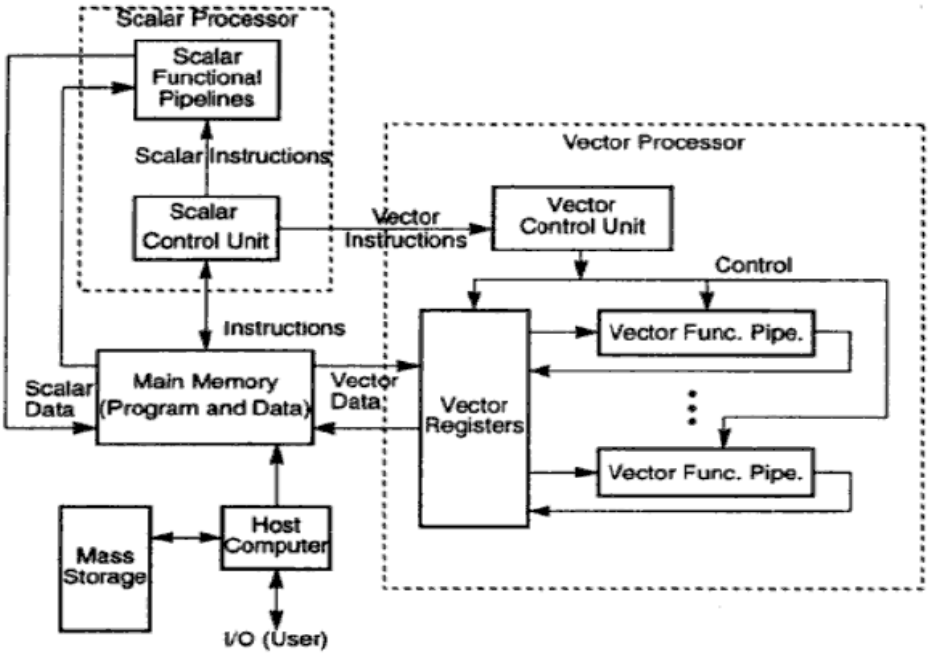


Figure 1.8: Architecture of Vector Supercomputer

SIMD Computer

The Synchronous parallel architectures coordinate Concurrent operations in lockstep through global clocks, central control units, or vector unit controllers. A synchronous array of parallel processors is called an array processor. These processors are composed of N identical processing elements (PES) under the supervision of a one control unit (CU) This Control unit is a computer with high speed registers, local memory and arithmetic logic unit.. An array processor is basically a single instruction and multiple data (SIMD) computers. There are N data streams; one per processor, so different data can be used in each processor.

The figure below show a typical SIMD or array processor

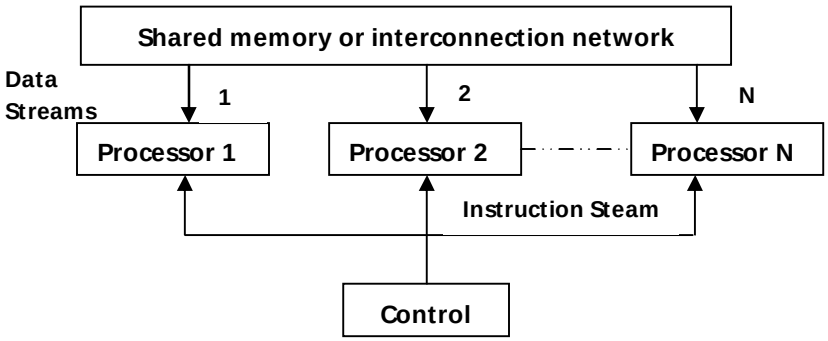


Figure 1.9: Configurations of SIMD Computers

These processors consist of a number of memory modules which can be either global or dedicated to each processor. Thus the main memory is the aggregate of the memory modules. These Processing elements and memory unit communicate with each other through an interconnection network. SIMD processors are especially designed for performing vector computations. SIMD has two basic architectural organizations

a. Array processor using random access memory

b. Associative processors using content addressable memory.

All N identical processors operate under the control of a single instruction stream issued by a central control unit. The popular examples of this type of SIMD configuration is ILLIAC IV, CM-2, MP-1. Each PEi is essentially an arithmetic logic unit (ALU) with attached working registers and local memory PEMi for the storage of distributed data. The CU also has its own main memory for the storage of program. The function of CU is to decode the instructions and determine where the decoded instruction should be executed. The PE perform same function (same instruction) synchronously in a lock step fashion under command of CU.

Data and Resource Dependence

Data dependence: The ordering relationship between statements is indicated by the data dependence. Five type of data dependence are defined below:

1. Flow dependence: A statement S2 is flow dependent on S1 if an execution path exists from s1 to S2 and if at least one output (variables assigned) of S1feeds in as input (operands to be used) to S2 also called RAW hazard and denoted as S1 -> S2

2. Anti-dependence: Statement S2 is anti-dependent on the statement S1 if S2 follows S1 in the program order and if the output of S2 overlaps the input to S1 also called RAW hazard and denoted as S1 | ->S2

3. Output dependence: two statements are output dependent if they produce (write) the same output variable. Also called WAW hazard and denoted as S1 0->S2

4. I/O dependence: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file referenced by both I/O statement.

5. Unknown dependence: The dependence relation between two statements cannot be determined in the following situations:

- The subscript of a variable is itself subscribed (indirect addressing).
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is non linear in the loop index variable.
- Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

Consider the following fragment of any program: S1 Load R1, A

S2 Add R2, R1

S3 Move R1, R3

S4 Store B, R1

- Here the Forward dependency S1to S2, S3 to S4, S2 to S2
- Anti-dependency from S2to S3
- Output dependency S1 toS3

Control Dependence: This refers to the situation where the order of the execution of statements cannot be determined before run time. For example all condition statement, where the flow of statement depends on the output. Different paths taken after a conditional branch may depend on the data hence we need to eliminate this data dependence among the instructions. This dependence also exists between operations performed in successive iterations of looping procedure. Control dependence often prohibits parallelism from being exploited.

Control-independent example:

```
for (i=0;i<n;i++) {
a[i] = c[i];
if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example:

```
for (i=1;i<n;i++) {
if (a[i-1] < 0) a[i] = 1;
}
```

Control dependence also avoids parallelism to being exploited. Compilers are used to eliminate this control dependence and exploit the parallelism.

Resource dependence:

Data and control dependencies are based on the independence of the work to be done. Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc. ALU conflicts are called ALU dependence. Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions - 1

Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

Notation

Ii is the set of all input variables for a process Pi. Ii is also called the read set or domain of Pi. Oi is the set of all output variables for a process Pi .Oi is also called write set

If P1 and P2 can execute in parallel (which is written as P1 || P2), then:

$$\mathbf{I}_1 \cap \mathbf{O}_2 = \varnothing$$
$$\mathbf{I}_2 \cap \mathbf{O}_1 = \varnothing$$
$$\mathbf{O}_1 \cap \mathbf{O}_2 = \varnothing$$

Bernstein's Conditions - 2

In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output- independent. The parallelism relation || is commutative (Pi || Pj implies Pj || Pi), but not transitive (Pi || Pj and Pj || Pk does not imply Pi || Pk). Therefore, || is not an equivalence relation. Intersection of the input sets is allowed.

Hardware and Software Parallelism

Hardware parallelism is defined by machine architecture and hardware multiplicity i.e., functional parallelism times the processor parallelism .It can be characterized by the number of instructions that can be issued per machine cycle. If a processor issues k instructions per machine cycle, it is called a k-issue processor. Conventional processors are one-issue machines. This provides the user the information about peak attainable performance.

Software Parallelism

Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph i.e., it is defined by dependencies within the code and is a function of algorithm, programming style, and compiler optimization.

Program partitioning and scheduling

Scheduling and allocation is a highly important issue since an inappropriate scheduling of tasks can fail to exploit the true potential of the system and can offset the gain from parallelization. In this paper we focus on the scheduling aspect. The objective of scheduling is to minimize the completion time of a parallel application by properly allocating the tasks to the processors. In a broad sense, the scheduling problem exists in two forms: static and dynamic. In static scheduling, which is usually done at compile time, the characteristics of a parallel program (such as task processing times, communication, data dependencies, and synchronization requirements) are known before program execution.

A parallel program, therefore, can be represented by a node- and edge-weighted directed acyclic graph (DAG), in which the node weights represent task processing times and the edge weights represent data dependencies as well as the communication times between tasks. In dynamic scheduling only, a few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made on-the-fly. The goal of a dynamic scheduling algorithm as such includes not only the minimization of the program completion time but also the minimization of the scheduling overhead which constitutes a significant portion of the cost paid for running the scheduler. In general dynamic scheduling is an NP hard problem.

Grain size and latency

The size of the parts or pieces of a program that can be considered for parallel execution can vary. The sizes are roughly classified using the term "granule size," or simply "granularity." The simplest measure, for example, is the number of instructions in a program part. Grain sizes are usually described as fine, medium or coarse, depending on the level of parallelism involved.

Latency

Latency is the time required for communication between different subsystems in a computer. Memory latency, for example, is the time required by a processor to access memory. Synchronization latency is the time required for two processes to synchronize their execution. Computational granularity and communication latency are closely related. Latency and grain size are interrelated and some general observation is

- As grain size decreases, potential parallelism increases, and overhead also increases.
- Overhead is the cost of parallelizing a task. The principle overhead is communication latency.
- As grain size is reduced, there are fewer operations between communications, and hence the impact of latency increases.
- Surface to volume: inter to intra-node communication.

Levels of Parallelism

Instruction Level Parallelism

This fine-grained or smallest granularity level typically involves less than 20 instructions per grain. The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.

Advantages:

- There are usually many candidates for parallel execution
- Compilers can usually do a reasonable job of finding this parallelism

Loop-level Parallelism

Typical loop has less than 500 instructions. If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine. Most optimized program construct to execute on a

parallel or vector machine. Some loops (e.g. recursive) are difficult to handle. Loop-level parallelism is still considered fine grain computation.

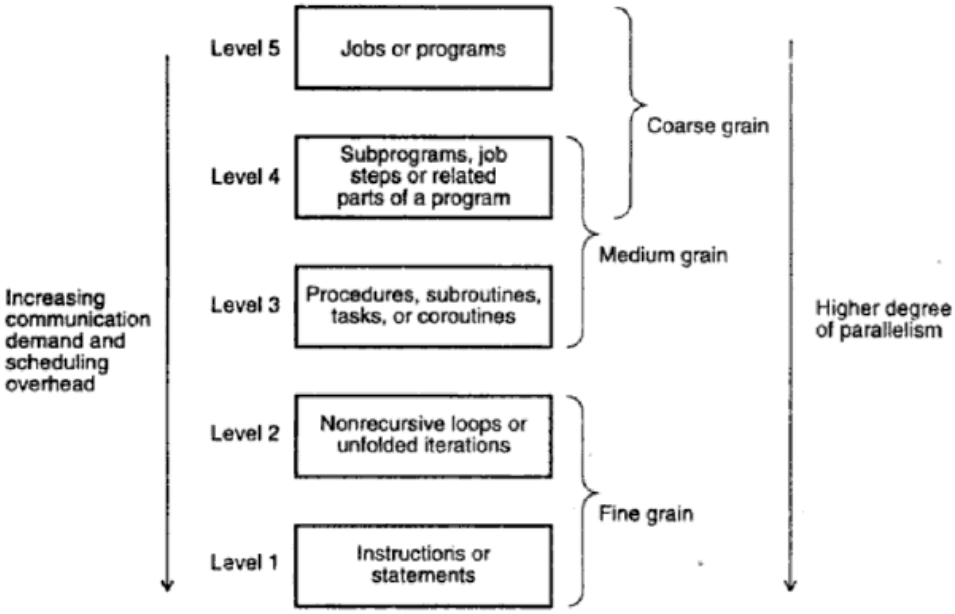


Figure 1.10: Level of Parallelism in Program Execution on Modern Computers

Procedure-level Parallelism

Medium-sized grain, usually less than 2000 instructions. Detection of parallelism is more difficult than with smaller grains; inter-procedural dependence analysis is difficult and history-sensitive. Communication requirement less than instruction level SPMD (single procedure multiple data) is a special case Multitasking belongs to this level.

Subprogram-level Parallelism

Job step level; grain typically has thousands of instructions; medium- or coarse-grain level. Job steps can overlap across different jobs. Multi-programming conducted at this level No compilers available to exploit medium- or coarse-grain parallelism at present.

Job or Program-Level Parallelism

It corresponds to execution of essentially independent jobs or programs on a parallel computer. This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

Balancing granularity and latency can yield better performance. Various latencies attributed to machine architecture, technology, and communication patterns used. Latency imposes a limiting factor on machine scalability. Ex. Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

Inter-processor Communication Latency

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns Ex. n communicating tasks may require n (n - 1)/2 communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include permutations broadcast multicast conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

Program Graphs and Packing

A program graph is similar to a dependence graph Nodes = { (n,s) }, where n = node name, s = size (larger

s = larger grain size).
Edges = { (v,d) }, where v = variable being "communicated," and d = communication delay.
Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.
Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

Scheduling

A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time. Some general scheduling goals:

- Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
- Select grain sizes for packing to achieve better schedules for a particular parallel machine.
- Node Duplication

Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule. By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.

Program flow mechanism

Conventional machines used control flow mechanism in which order of program execution explicitly stated in user programs. Dataflow machines which instructions can be executed by determining operand availability.
Reduction machines trigger an instruction's execution based on the demand for its results.
Control Flow vs. Data Flow: In Control flow computers the next instruction is executed when the last instruction as stored in the program has been executed where as in Data flow computers an instruction executed when the data (operands) required for executing that instruction is available Control flow machines used shared memory for instructions and data. Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing. Single processor systems are inherently sequential.
Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. Data tokens are passed from an instruction to its dependents to trigger execution.

Data Flow Features

No need for shared memory program counter control sequencer Special mechanisms are required to detect data availability match data tokens with instructions needing them enable chain reaction of asynchronous instruction execution
A Dataflow Architecture – 1 The Arvind machine (MIT) has N PEs and an N -by –N interconnection network. Each PE has a token-matching mechanism that dispatches only instructions with data tokens available. Each datum is tagged with

- address of instruction to which it belongs
- context in which the instruction is being executed

Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network. Instruction addresses effectively replace the program counter in a control flow machine. Context identifier effectively replaces the frame base register in a control flow machine. Since the dataflow machine matches the data tags from one instruction with successors, synchronized instruction execution is implicit.
An I-structure in each PE is provided to eliminate excessive copying of data structures. Each word of the I-structure has a two-bit tag indicating whether the value is empty, full, or has pending read requests.
This is a retreat from the pure dataflow approach. Special compiler technology needed for dataflow machines.

Demand-Driven Mechanisms

Data-driven machines select instructions for execution based on the availability of their operands; this is essentially a bottom-up approach.

Demand-driven machines take a top-down approach, attempting to execute the instruction (a demander) that yields the final result. This triggers the execution of instructions that yield its operands, and so forth. The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).

Pattern driven computers: An instruction is executed when we obtain a particular data patterns as output. There are two types of pattern driven computers

String-reduction model: each demander gets a separate copy of the expression string to evaluate each reduction step has an operator and embedded reference to demand the corresponding operands each operator is suspended while arguments are evaluated

Graph-reduction model: expression graph reduced by evaluation of branches or sub-graphs, possibly in parallel, with demanders given pointers to results of reductions. Based on sharing of pointers to arguments; traversal and reversal of pointers continues until constant arguments are encountered.

System interconnect architecture

Various types of interconnection networks have been suggested for SIMD computers. These are basically classified have been classified on network topologies into two categories namely

- Static Networks
- Dynamic Networks

Static versus Dynamic Networks

The topological structure of an SIMD array processor is mainly characterized by the data routing network used in interconnecting the processing elements.

The topological structure of an SIMD array processor is mainly characterized by the data routing network used in the interconnecting the processing elements. To execute the communication the routing function f is executed and via the interconnection network the PEi copies the content of its Ri register into the Rf(i) register of PE f(i). The f(i) the processor identified by the mapping function f. The data routing operation occurs in all active PEs simultaneously.

Network properties and routing

The goals of an interconnection network are to provide low-latency high data transfer rate wide communication bandwidth. Analysis includes latency bisection bandwidth data- routing functions scalability of parallel architecture These Network usually represented by a graph with a finite number of nodes linked by directed or undirected edges.

Number of nodes in graph = network size.

Number of edges (links or channels) incident on a node = node degree d (also note in and out degrees when edges are directed).

Node degree reflects number of I/O ports associated with a node, and should ideally be small and constant.

Network is symmetric if the topology is the same looking from any node; these are easier to implement or to program.

Diameter: The maximum distance between any two processors in the network or in other words we can say Diameter, is the maximum number of (routing) processors through which a message must pass on its way from source to reach destination. Thus diameter measures the maximum delay for transmitting a message from one processor to another as it determines communication time hence smaller the diameter better will be the network topology.

Connectivity: How many paths are possible between any two processors i.e., the multiplicity of paths between two processors. Higher connectivity is desirable as it minimizes contention.
Arch connectivity of the network: the minimum number of arcs that must be removed for the network to break it into two disconnected networks. The arch connectivity of various network are as follows

- 1 for linear arrays and binary trees
- 2 for rings and 2-d meshes
- 4 for 2-d torus
- d for d-dimensional hypercubes

Larger the arch connectivity lesser the conjunctions and better will be network topology. Channel width :The channel width is the number of bits that can communicated simultaneously by a interconnection bus connecting two processors:
Bisection Width and Bandwidth: In order divide the network into equal halves we require the remove some communication links. The minimum numbers of such communication links that have to be removed are called the Bisection Width. Bisection width basically provide us the information about the largest number of messages which can be sent simultaneously (without needing to use the same wire or routing processor at the same time and so delaying one another), no matter which processors are sending to which other processors. Thus larger the bisection width is the better the network topology is considered. Bisection Bandwidth is the minimum volume of communication allowed between two halves of the network with equal numbers of processors. This is important for the networks with weighted arcs where the weights correspond to the link width i.e., (how much data it can transfer). The Larger bisection width the better network topology is considered.
Cost the cost of networking can be estimated on variety of criteria where we consider the number of communication links or wires used to design the network as the basis of cost estimation, smaller the better the cost.
Data Routing Functions: A data routing network is used for inter –PE data exchange. It can be static as in case of hypercube routing network or dynamic such as multistage network. Various type of data routing functions are Shifting, Rotating, Permutation (one to one), Broadcast (one to all), Multicast (many to many), Personalized broadcast (one to many), Shuffle, Exchange Etc.

Factors Affecting Performance
Functionality – how the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence.
Network latency – worst-case time for a unit message to be transferred
Bandwidth – maximum data rate.
Hardware complexity – implementation costs for wire, logic, switches, connectors, etc. Scalability – how easily does the scheme adapt to an increasing number of processors, memories, etc.

Static interconnection networks
Static interconnection networks for elements of parallel systems (ex. processors, memories) are based on fixed connections that cannot be modified without a physical re-designing of a system. Static interconnection networks can have many structures such as a linear structure (pipeline), a matrix, a ring, a torus, a complete connection structure, a tree, a star, a hyper-cube.

In linear and matrix structures, processors are interconnected with their neighbors in a regular structure on a plane. A torus is a matrix structure in which elements at the matrix borders are connected in the frame of the same lines and columns. In a complete connection structure, all elements (ex. processors) are directly interconnected (point-to-point)

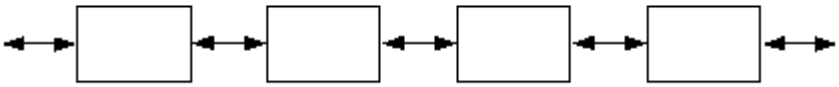


Figure 1.11: Linear structure (pipeline) of interconnections in a parallel system

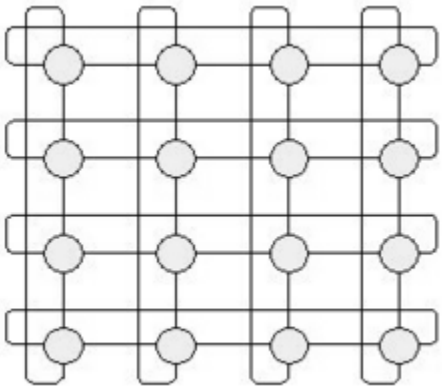


Figure 1.12: 2D Torus

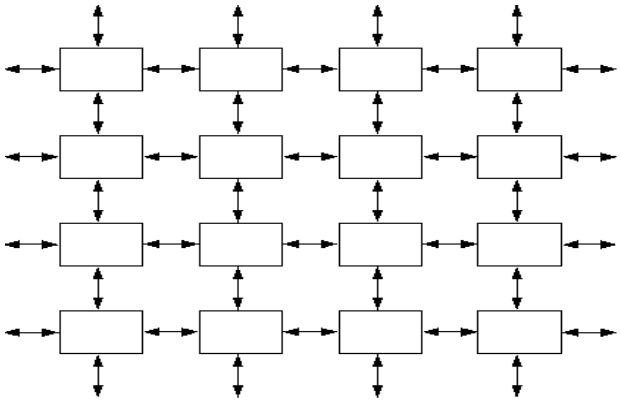


Figure 1.13: M atrix

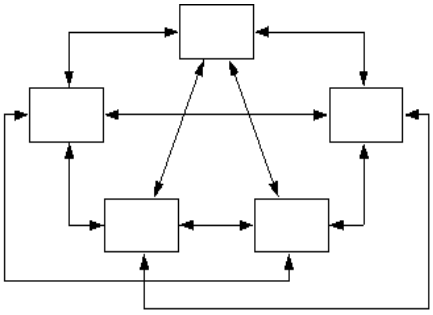


Figure 1.14: A complete interconnection

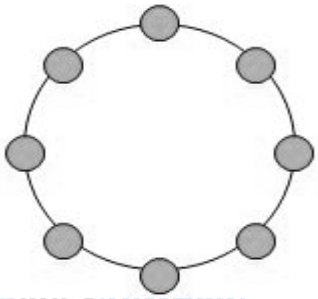


Figure 1.15: A Ring

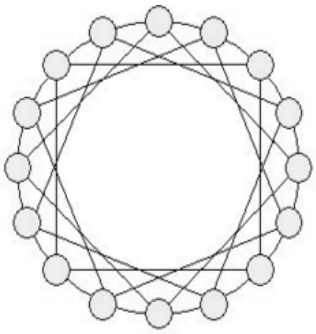


Figure 1.16: A Chordal Ring

In a tree structure, system elements are set in a hierarchical structure from the root to the leaves, see the figure below. All elements of the tree (nodes) can be processors or only leaves are processors and the rest of nodes are linking elements, which intermediate in transmissions. If from one node, 2 or more connections go to different nodes towards the leaves - we say about a binary or k-nary tree. If from one node, more than one connection goes to the neighboring node, we speak about a fat tree. A binary tree, in which in the direction of the root, the number of connections between neighboring nodes increases twice, provides a uniform transmission throughput between the tree levels, a feature not available in a standard tree.

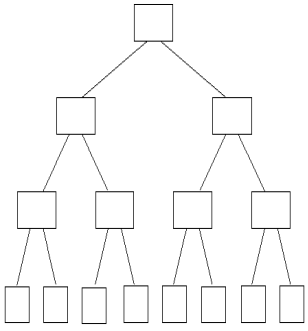


Figure 1.17: Binary tree

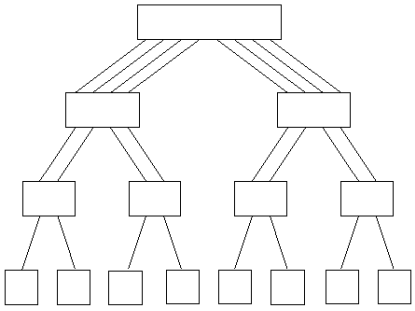


Figure 1.18: Fat tree

In a hypercube structure, processors are interconnected in a network, in which connections between processors correspond to edges of an n-dimensional cube. The hypercube structure is very advantageous since it provides a low network diameter equal to the degree of the cube. The network diameter is the number of edges between the most distant nodes. . The network diameter determines the number in intermediate

transfers that have to be done to send data between the most distant nodes of a network. In this respect the hypercubes have very good properties, especially for a very large number of constituent nodes. Due to this hypercubes are popular networks in existing parallel systems.

Dynamic interconnection networks

Dynamic interconnection networks between processors enable changing (reconfiguring) of the connection structure in a system. It can be done before or during parallel program execution. So, we can speak about static or dynamic connection reconfiguration.

The dynamic networks are those networks where the route through which data move from one PE to another is established at the time communication has to be performed. Usually all processing elements are equidistant and an interconnection path is established when two processing element want to communicate by use of switches. Such systems are more difficult to expand as compared to static network. Examples: Bus-based, Crossbar, Multistage Networks. Here the Routing is done by comparing the bit-level representation of source and destination addresses. If there is a match goes to next stage via pass- through else in case of it mismatch goes via cross-over using the switch.

There are two classes of dynamic networks namely

- single stage network
- multi stage

Single Stage Networks

A single stage switching network with N input selectors (IS) and N output selectors (OS). Here at each network stage there is a 1- to-D demultiplexer corresponding to each IS such that 1<D<N and each OS is an M-to-1 multiplexer such that 1<M<=N. Cross bar network is a single stage network with D=M=N. In order to establish a desired connecting path different path control signals will be applied to all IS and OS selectors. The single stage network is also called as re-circulating network as in this network connection the single data items may have to re-circulate several time through the single stage before reaching their final destinations. The number of recirculation depends on the connectivity in the single stage network. In general higher the hardware connectivity the lesser is the number of recirculation. In cross bar network only one circulation is needed to establish the connection path. The cost of completed connected cross bar network is O(N2) which is very high as compared to other most re-circulating networks which have cost O(N log N) or lower hence are more cost effective for large value of N.

Multistage Networks

Many stages of interconnected switches form a multistage SIMD network. It is basicaaly consist of three characteristic features

- The switch box,
- The network topology
- The control structure

Many stages of interconnected switches form a multistage SIMD networks. Eachbox is essentially an interchange device with two inputs and two outputs. The four possible states of a switch box are which are shown in figure.

- Straight
- Exchange
- Upper Broadcast
- Lower broadcast.

A two function switch can assume only two possible state namely state or exchange states. However a four function switch box can be any of four possible states. A multistage network is capable of connecting any input terminal to any output terminal. Multi-stage networks are basically constructed by so called shuffle-exchange switching element, which is basically a 2 x 2 crossbar. Multiple layers of these elements are connected and form the network.

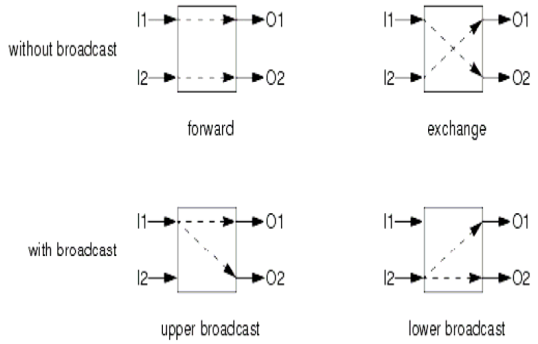


Figure 1.19: A two-by-two switching box and its four interconnection states

A multistage network is capable of connecting an arbitrary input terminal to an arbitrary output terminal. Generally it is consist of n stages where $N = 2^n$ is the number of input and output lines. And each stage use N/2 switch boxes. The interconnection patterns from one stage to another stage are determined by network topology. Each stage is connected to the next stage by at least N paths. The total wait time is proportional to the number stages i.e., n and the total cost depend on the total number of switches used and that are $N\log_2 N$. The control structure can be individual stage control i.e., the same control signal is used to set all switch boxes in the same stages thus we need n control signal. The second control structure is individual box control where a separate control signal is used to set the state of each switch box. This provide flexibility at the same time require n2/2 control signal which increases the complexity of the control circuit. In between path is use of partial stage control.

Bus networks

A bus is the simplest type of dynamic interconnection networks. It constitutes a common data transfer path for many devices. Depending on the type of implemented transmissions we have serial busses and parallel busses. The devices connected to a bus can be processors, memories, I/O units, as shown in the figure below.

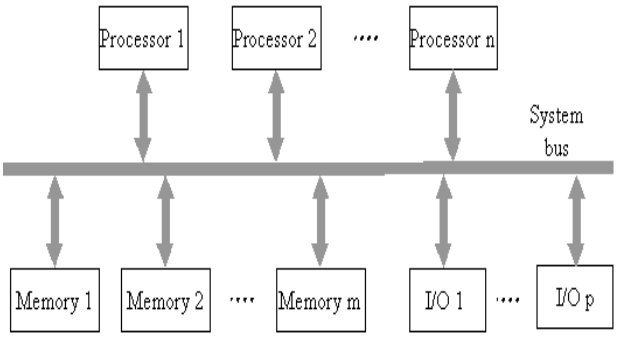


Figure 1.20: A diagram of a system based on a single bus

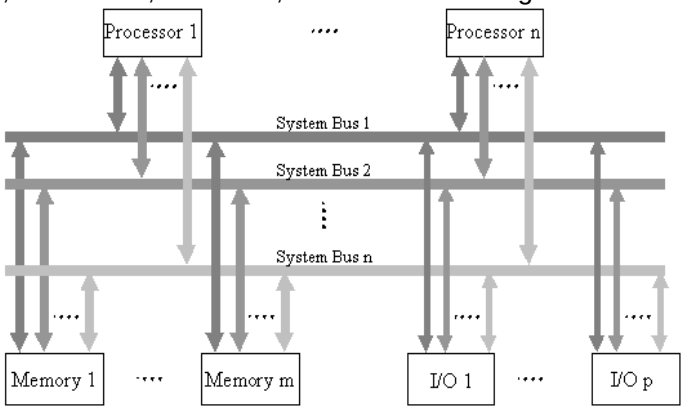


Figure 1.21: A diagram of a system based on a multibus

Only one device connected to a bus can transmits data. Many devices can receive data. In the last case we speak about a multicast transmission. If data are meant for all devices connected to a bus we speak about a broadcast transmission. Accessing the bus must be synchronized. It is done with the use of two methods: a token method and a bus arbiter method. With the token method, a token (a special control message or signal) is circulating between the devices connected to a bus and it gives the right to transmit to the bus to a single device at a time. The bus arbiter receives data transmission requests from the devices connected to a bus. It selects one device according to a selected strategy (ex. using a system of assigned priorities) and sends an acknowledge message (signal) to one of the requesting devices that grants it the transmitting right. After the selected device completes the transmission, it informs the arbiter that can select another request. The receiver (s) address is usually given in the header of the message. Special header values are used for the broadcast and multicasts. All receivers read and decode headers. These devices that are specified in the header, read-in the data transmitted over the bus.

The throughput of the network based on a bus can be increased by the use of a multibus network shown in the figure below. In this network, processors connected to the busses can transmit data in parallel (one for each bus) and many processors can read data from many busses at a time.

Crossbar switches

A crossbar switch is a circuit that enables many interconnections between elements of a parallel system at a time. A crossbar switch has a number of input and output data pins and a number of control pins. In response to control instructions set to its control input, the crossbar switch implements a stable connection of a determined input with a determined output. The diagrams of a typical crossbar switch are shown in the figure below.

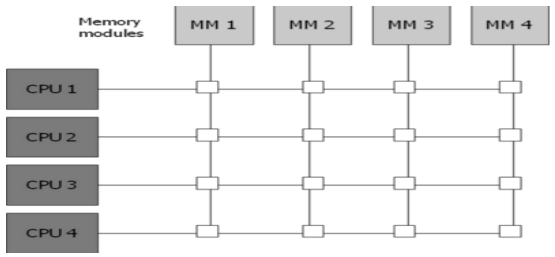


Figure 1.22: Crossbar switch

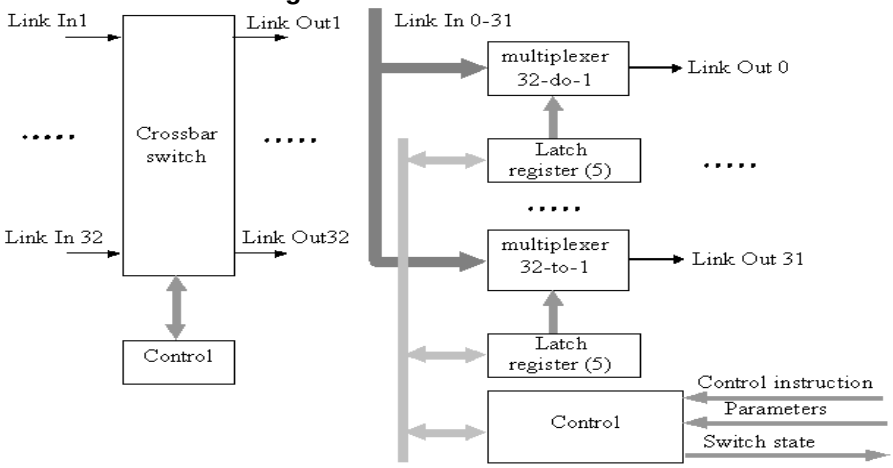


Figure 1.23: Crossbar switch a) general scheme, b) internal structure

Control instructions can request reading the state of specified input and output pins i.e. their current connections in a crossbar switch. Crossbar switches are built with the use of multiplexer circuits, controlled by latch registers, which are set by control instructions. Crossbar switches implement direct, single non-blocking connections, but on the condition that the necessary input and output pins of the switch are free. The connections between free pins can always be implemented independently on the status of other connections. New connections can be set during data transmissions through other connections. The non-blocking connections are a big advantage of crossbar switches. Some crossbar switches enable broadcast transmissions but in a blocking manner for all other connections. The disadvantage of crossbar switches is that extending their size, in the sense of the number of input/output pins, is costly in terms of hardware. Because of that, crossbar switches are built up to the size of 100 input/output pins.

Multiport Memory

In the multiport memory system, different memory module and CPUs have separate buses. The module has internal control logic to determine port which will access to memory at any given time. Priorities are assigned to each memory port to resolve memory access conflicts.

Advantages:

Because of the multiple paths high transfer rate can be achieved.

Disadvantages:

It requires expensive memory control logic and a large number of cables and connections.

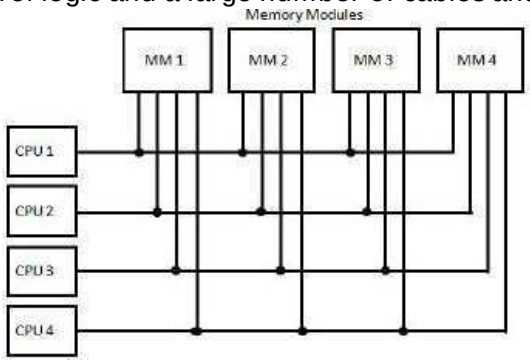


Figure 1.24: Multiport memory organization

Multistage and combining networks

Multistage connection networks are designed with the use of small elementary crossbar switches (usually they have two inputs) connected in multiple layers. The elementary crossbar switches can implement 4 types of connections: straight, crossed, upper broadcast and lower broadcast. All elementary switches are controlled simultaneously. The network like this is an alternative for crossbar switches if we have to switch a large number of connections, over 100. The extension cost for such a network is relatively low.

In such networks, there is no full freedom in implementing arbitrary connections when some connections have already been set in the switch. Because of this property, these networks belong to the category of so called blocking networks.

However, if we increase the number of levels of elementary crossbar switches above the number necessary to implement connections for all pairs of inputs and outputs, it is possible to implement all requested connections at the same time but statically, before any communication is started in the switch. It can be achieved at the cost of additional redundant hardware included into the switch. The block diagram of such a network, called the Benes network, is shown in the figure below.

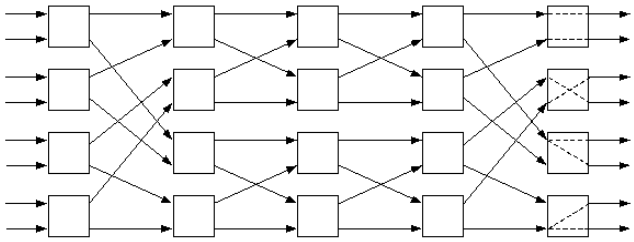


Figure 1.25: A multistage connection network for parallel systems

To obtain nonblocking properties of the multistage connection network, the redundancy level in the circuit should be much increased. To build a nonblocking multistage network n x n, the elementary two-input switches have to be replaced by 3 layers of switches n x m, r x r and m x n, where m ³ 2n - 1 and r is the number of elementary switches in the layer 1 and 3. Such a switch was designed by a French mathematician Clos and it is called the Clos network. This switch is commonly used to build large integrated crossbar switches. The block diagram of the Clos network is shown in the figure below.

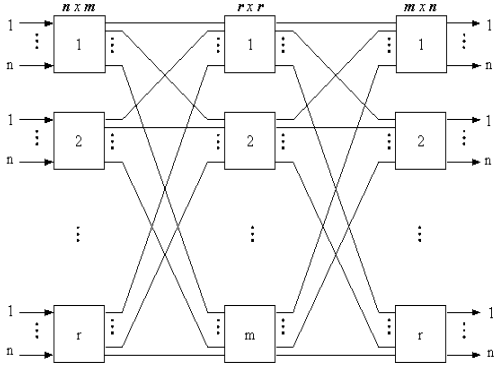


Figure 1.26: A nonblocking Clos interconnection network

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in

RGPVNOTES.IN

Subject Name: **Advanced Computer Architecture**

Subject Code: **CS-6001**

Semester: **6th**

Department of Computer Science and Engineering

Subject Notes

Subject Code: CS-6001 **Subject Name: Advanced Computer Architecture**

UNIT-2

Instruction Set Architectures

The instruction set, also called instruction set architecture (ISA), is part of a computer that pertains to programming, which is basically machine language. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.

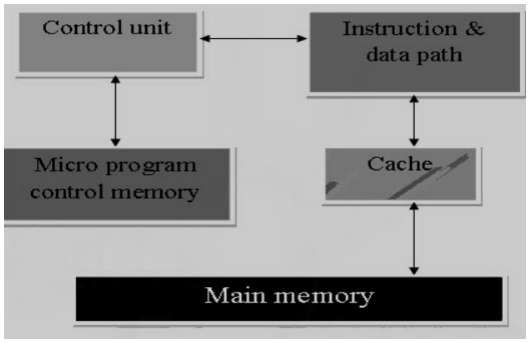
Examples of instruction set

- ADD - Add two numbers together.
- COMPARE - Compare numbers.
- IN - Input information from a device, e.g., keyboard.
- JUMP - Jump to designated RAM address.
- LOAD - Load information from RAM to the CPU.
- OUT - Output information to device, e.g., monitor.
- STORE - Store information to RAM.

Computers are classified on the basis on instruction set they have as:

CISC Scalar Processors

CISC (Complex Instruction Set Computer): CISC based computer will have shorter programs which are made up of symbolic machine language. A Complex Instruction Set Computer (CISC) supplies a large number of complex instructions at the assembly language level. During the early years, memory was slow and expensive and the programming was done in assembly language. Since memory was slow and instructions could be retrieved up to 10 times faster from a local ROM than from main memory, programmers tried to put as many instructions as possible in a microcode.



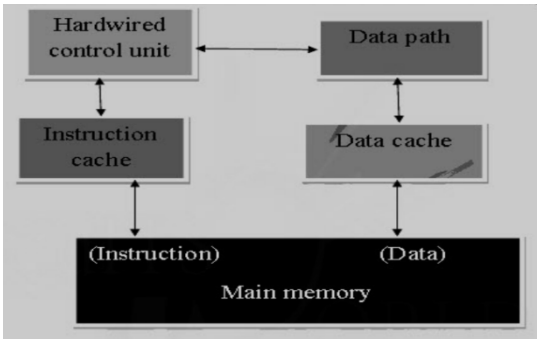


Figure 2.1: (a) CSIC Architecture **(b) RSIC Architecture**

RISC Scalar Processors

RISC (Reduced Instruction Set Computer): RISC is a type of microprocessor that has a relatively limited number of instructions. It is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed (perform more million instructions per second, or millions of instructions per second). Earlier, computers used only 20% of the instructions, making the other 80% unnecessary. One advantage of reduced instruction set computers is that they can execute their instructions very fast because the instructions are so simple.

RISC chips require fewer transistors, which makes them cheaper to design and produce. In a RISC machine, the instruction set contains simple, basic instructions, from which more complex instructions can be composed. Each instruction is of the same length, so that it may be fetched in a single operation. Most instructions complete in one machine cycle, which allows the processor to handle several instructions at the same time. This pipelining is a key technique used to speed up RISC machines.

Advantages:

- Speed: Since a simplified instruction set allows for a pipelined, superscalar design RISC processors often achieve 2 to 4 times the performance of CISC processor using comparable semiconductor technology and the same clock rates.
- Simpler Hardware : Because the instruction set of a RISC processor is so simple, it uses up much less chip space; extra functions, such as memory management units or floating point arithmetic units, can also be placed on the same chip. Smaller chips allow a semiconductor manufacturer to place more parts on a single silicon wafer, which can lower the per-chip cost dramatically.
- Shorter Design Cycle : Since RISC processors are simpler than corresponding CISC processors, they can be designed more quickly, and can take advantage of other technological developments sooner than corresponding CISC designs, leading to greater leaps in performance between generations.

Difference between CISC and RISC

Architectural Characterstics	Complex Instruction Set Computer(CISC)	Reduced Instruction Set Computer(RISC)
Instruction size and format	Large set of instructions with variable formats (16-64 bits per instruction).	Small set of instructions with fixed format (32 bit).
Data transfer	Memory to memory.	Register to register.
CPU control	Most micro coded using control memory (ROM) but modern CISC use hardwired control.	Mostly hardwired without control memory.
Instruction type	Not register based instructions.	Register based instructions.
Memory access	More memory access.	Less memory access.
Clocks	Includes multi-clocks.	Includes single clock.
Instruction nature	Instructions are complex.	Instructions are reduced and simple.

VLIW Architecture

Very long instruction word (VLIW) describes a computer processing architecture in which a language compiler or pre-processor breaks program instruction down into basic operations that can be performed by the processor in parallel (that is, at the same time). These operations are put into a very long instruction word which the processor can then take apart without further analysis, handing each operation to an appropriate functional unit.

VLIW is sometimes viewed as the next step beyond the reduced instruction set computing (RISC) architecture, which also works with a limited set of relatively basic instructions and can usually execute more than one instruction at a time (a characteristic referred to as superscalar). The main advantage of VLIW processors is that complexity is moved from the hardware to the software, which means that the hardware can be smaller, cheaper, and require less power to operate. The challenge is to design a compiler or pre-processor that is intelligent enough to decide how to build the very long instruction words. If dynamic pre-processing is done as the program is run, performance may be a concern.

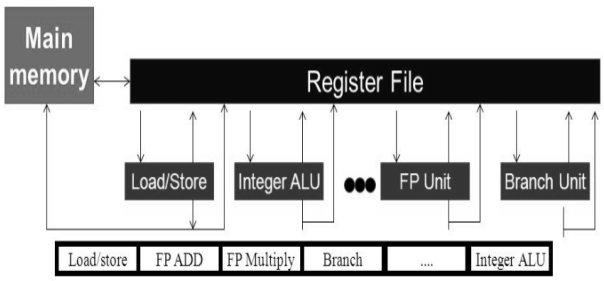


Figure 2.2: A VLIW processor architecture and instruction format

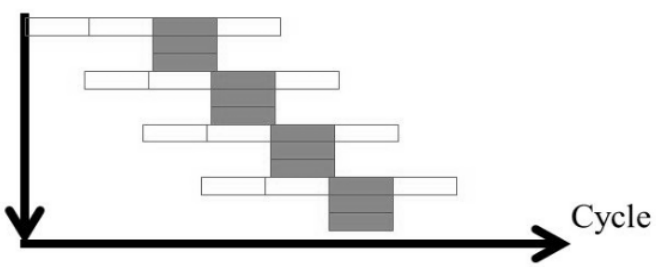


Figure 2.3: Pipeline execution

Pipelining in VLIW Processors

Decoding of instructions is easier in VLIW than in superscalars, because each "region" of an instruction word is usually limited as to the type of instruction it can contain.

Code density in VLIW is less than in superscalars, because if a "region" of a VLIW word isn't needed in a particular instruction, it must still exist (to be filled with a "no op"). Superscalars can be compatible with scalar processors; this is difficult with VLIW parallel and non-parallel architectures.

VLIW Opportunities

"Random" parallelism among scalar operations is exploited in VLIW, instead of regular parallelism in a vector or SIMD machine.

The efficiency of the machine is entirely dictated by the success, or "goodness," of the compiler in planning the operations to be placed in the same instruction words.

Different implementations of the same VLIW architecture may not be binary-compatible with each other, resulting in different latencies.

Memory Hierarchy

The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory. Auxiliary memory access time is generally 1000 times that of the main memory, hence it is at the bottom of the hierarchy.

The main memory occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.

The cache memory is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about 1 to 7~10

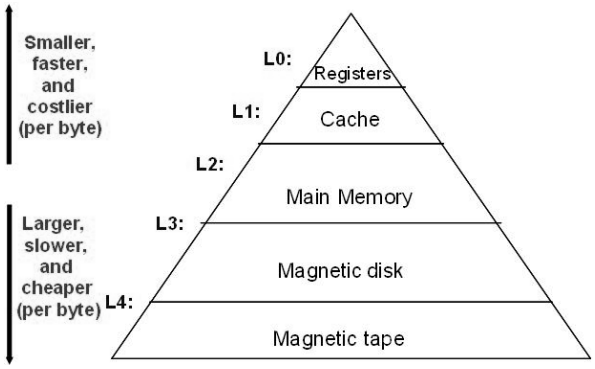


Figure 2.4: Memory Hierarchy

1. Internal register: Internal register in a CPU is used for holding variables and temporary results. Internal registers have a very small storage; however they can be accessed instantly. Accessing data from the internal register is the fastest way to access memory.
2. Cache: Cache is used by the CPU for memory which is being accessed over and over again. Instead of pulling it every time from the main memory, it is put in cache for fast access. It is also a smaller memory, however, larger than internal register.
   Cache is further classified to L1, L2 and L3:
   a) L1 cache: It is accessed without any delay.
   b) L2 cache: It takes more clock cycles to access than L1 cache.
   c) L3 cache: It takes more clock cycles to access than L2 cache.

3. Main memory or RAM (Random Access Memory): It is a type of the computer memory and is a hardware component. It can be increased provided the operating system can handle it. Typical PCs these days use 8 GB of RAM. It is accessed slowly as compared to cache.
4. Hard disk: A hard disk is a hardware component in a computer. Data is kept permanently in this memory. Memory from hard disk is not directly accessed by the CPU, hence it is slower. As compared with RAM, hard disk is cheaper per bit.
5. Magnetic tape: Magnetic tape memory is usually used for backing up large data. When the system needs to access a tape, it is first mounted to access the data. When the data is accessed, it is then un-mounted. The memory access time is slower in magnetic tape and it usually takes few minutes to access a tape.

Memory Hierarchy Properties:

Information stored in a memory hierarchy (M1, M2,..Mn) satisfies three important properties:

Inclusion Property: it implies that all information items are originally stored in level Mn. During the processing, subsets

of Mn are copied into Mn-1 similarly, subsets of Mn-1 are copied into Mn-2, and so on.

Coherence Property: it requires that copies of the same information item at successive memory levels be consistent. If a word is modified in the cache, copies of that word must be updated immediately or eventually at all higher levels..

Locality of References: the memory hierarchy was developed based on a program behavior known as locality of references. Memory references are generated by the CPU for either instruction or data access. Frequently used information is found in the lower levels in order to minimize the effective access time of the memory hierarchy.

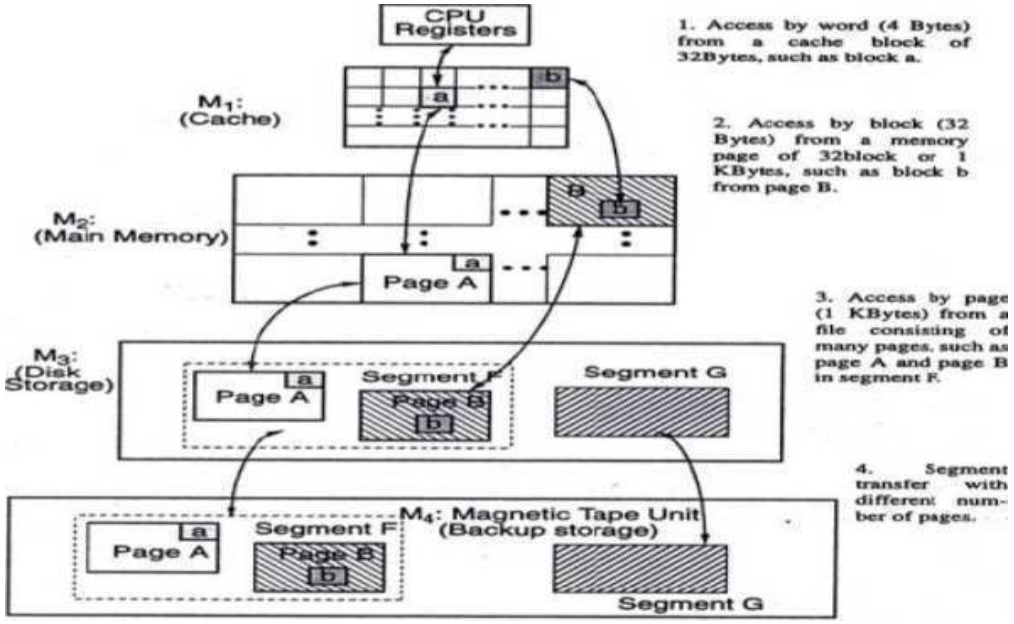


Figure 2.5: The inclusion property and data transfer between adjacent levels

The following three principles which led to an effective implementation memory hierarchy for a system are:

1 Make the Common Case Fast: This principle says the data which is more frequently used should be kept in faster device. It is based on a fundamental law, called Amdahl's Law, which states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Thus if faster mode use relatively less frequent data then most of the time faster mode device will not be used hence the speed up achieved will be less than if faster mode device is more frequently used.

2. Principle of Locality: It is very common trend of Programs to reuse data and instructions that are used

recently. Based on this observation comes important program property called locality of references: the instructions and data in a program that will be used in the near future is based on its accesses in the recent past. There is a famous **40/10 rule that** comes from empirical observation is:

"A program spends 40% of its time in 10% of its code"

These localities can be categorized of three types:

a. Temporal locality: states that data items and code that are recently accessed are likely to be accessed in the near future. Thus if location M is referenced at time t, then it (location M) will be referenced again at some time t+Dt.

b. Spatial locality: states that items try to reside in proximity in the memory i.e., the items whose addresses are near to each other are likely to be referred together in time. Thus we can say memory accesses are clustered with respect to the address space. Thus if location M is referenced at time t, then another location M±Dm will be referenced at time t+Dt.

c. Sequential locality: Programs are stored sequentially in memory and normally these programs has sequential trend of execution. Thus we say instructions are stored in memory in certain array patterns and are accessed sequentially one memory locations after another. Thus if location M is referenced at time t, then locations M+1, M+2, ...will be referenced at time t+Dt, t+Dt', etc. In each of these patterns, both Dm and Dt are "small." H&P suggest that 90 percent of the execution time in most programs is spent executing only 10 percent of the code. One of the implications of the locality is data and instructions should have separate data and instruction caches. The main advantage of separate caches is that one can fetch instructions and operands simultaneously. This concept is basis of the design known as Harvard architecture, after the Harvard Mark series of electromechanical machines, in which the instructions were supplied by a separate unit.

3. Smaller is Faster: Smaller pieces of hardware will generally be faster than larger pieces.

This according to above principles suggested that one should try to keep recently accessed items in the fastest memory.

While designing the memory hierarchy following points are always considered

Inclusion property: If a value is found at one level, it should be present at all of the levels below it.

$$M_1 \subset M_2 \subset \ldots \subset M_n$$

The implication of the inclusion property is that all items of information in the "innermost" memory level (cache) also appear in the outer memory levels.The inverse, however, is not necessarily true. That is, the presence of a data item in level Mi+1 does not imply its presence in level Mi. We call a reference to a missing item a "miss."

The Coherence Property

The value of any data should be consistent at all level. The inclusion property is, of course, never completely true, but it does represent a desired state. That is, as information is modified by the processor, copies of that information should be placed in the appropriate locations in outer memory levels. The requirement that copies of data items at successive memory levels be consistent is called the "coherence property."

Coherence Strategies

Write-through

As soon as a data item in M i is modified, immediate update of the corresponding data item(s) in M i+1, Mi+2, ...Mn is required. This is the most aggressive (and expensive) strategy.

Write-back

The update of the data item in M i+1 corresponding to a modified item in Mi is not updated unit it (or the block/page/etc. in M i that contains it) is replaced or removed. This is the most efficient approach, but cannot be used (without modification) when multiple processors share Mi+1, .., Mn.

Memory Capacity Planning:

The performance of a memory hierarchy is determined by the effective access time (Teff) to any level in the hierarchy. It depends on the hit ratio and access frequencies at successive levels.

Hit Ratio (h): is a concept defined for any two adjacent levels of a memory hierarchy. When an information item found in Mi, it is a hit, otherwise, a miss. The hit ratio (hi) at Mi is the probability that an information item will be found in Mi. the miss ratio at Mi is defined as 1-hi.

The **access frequency** to Mi is defined as

fi= (1-h1)(1-h2)….(1-hi)

Effective Access Time (Teff):

In practice, we wish to achieve as high a hit ratio as possible at M1. Every time a miss occurs, a penalty must be paid to access the next higher level of memory. The Teff of a memory hierarchy is given by:

$$\begin{aligned} T_{eff} &= \sum_{i=1}^{n} f_i \cdot t_i \\ &= h_1 t_1 + (1-h_1)h_2 t_2 + (1-h_1)(1-h_2)h_3 t_3 + \cdots + \\ &\quad (1-h_1)(1-h_2)\cdots(1-h_{n-1})t_n \end{aligned}$$

Hierarchy Optimization:

The total cost of a memory hierarchy is estimated as:

$$C_{total} = \sum_{i=1}^{n} c_i \cdot s_i$$

Interleaved memory organization- memory interleaving

It is a technique for compensating the relatively slow speed of DRAM (Dynamic RAM). In this technique, the main memory is divided into memory banks which can be accessed individually without any dependency on the other.

High-Order Interleaving

Arguably the most "natural" arrangement would be to use bus lines A26-A27 as the module determiner. In other words, we would feed these two lines into a 2-to-4 decoder, the outputs of which would be connected to the Chip Select pins of the four memory modules. If we were to do this, the physical placement of our system addresses would be as follows:

address	module
0-64M	0
64M-128M	1
128M-192M	2
192M-256M	3

Note that this means consecutive addresses are stored within the same module, except at the boundary. The above arrangement is called high-order interleaving, because it uses the high-order, i.e. most significant, bits of the address to determine which module the word is stored in.

Low-Order Interleaving

An alternative would be to use the low bits for that purpose. In our example here, for instance, this would entail feeding bus lines A0-A1 into the decoder, with bus lines A2-A27 being tied to the address pins of the memory modules. This would mean the following storage pattern:

address	module
0	0
1	1
2	2
3	3
4	0
5	1
6	2
7	3
8	0
9	1
etc.	etc.

In other words, consecutive addresses are stored in consecutive modules, with the understanding that this is mod 4, i.e. we wrap back to M0 after M3.

Bandwidth

The memory bandwidth (B) of an m-way interleaved memory is lower-bounded by 1 and upper-bounded by m. The approximation of B by Hellerman is:

$$B = m^{0.56} \sim \sqrt{m}$$

In this equation m denotes the number of interleaved memory modules. This equation indicated that the

efficient memory bandwidth is approximately two times that of single module when four memory modules are used.

This pessimistic estimate is because of the fact that block access of different lengths and access of single words are randomly mixed in users programs. Hellerman's calculation was depend on a single processor system. The effective memory bandwidth decreased again, if memory-access conflicts from multiple processors are considered.

Fault Tolerance

To achieve various interleaved memory organizations, low order and high order interleaving are combined. In each memory module, sequential addresses are allocated in high order interleaved memory. This makes it simple to isolate faulty memory modules in a memory bank of m memory modules. If one module failure is detected the remaining modules can still be used by opening the window in the address space. This fault isolation cannot be performed in low order interleaved memory, where a module failure may paralyze the complete memory bank. Hence, low order interleaved memory is not fault tolerant.

Backplane Buses

A backplane bus interconnects processors, data storage and peripheral devices in a tightly coupled hardware. The system bus must be designed to allow communication between devices on the devices on the bus without disturbing the internal activities of all the devices attached to the bus. These are typically `intermediate' buses, used to connect a variety of other buses to the CPU-Memory bus. They are called Backplane Buses because they are restricted to the backplane of the system.

Backplane bus specification

They are generally connected to the CPU-Memory bus by a bus adaptor, which handles translation between the buses. Commonly, this is integrated into the CPU-Memory bus controller logic. While these buses can be used to directly control devices, they are used as 'bridges` to other buses. For example, AGP bus devices – i.e. video cards – act as bridges between the CPU-Memory bus and the actual display device: the monitor.

- Allow processors, memory and I/O devices to coexist on single bus.
- Balance demands of processor-memory communication with demands of I/O device-memory communication.
- Interconnects the circuit boards containing processor, memory and I/O interfaces an interconnection structure within the chassis.
- Data address and control lines form the data transfer bus (DTB) in VME bus.
- DTB Arbitration bus that provide control of DTB to requester using the arbitration logic.
- Interrupt and Synchronization bus used for handling interrupt.
- Utility bus includes signals that provide periodic timing and coordinate the power up and power down sequence of the system.

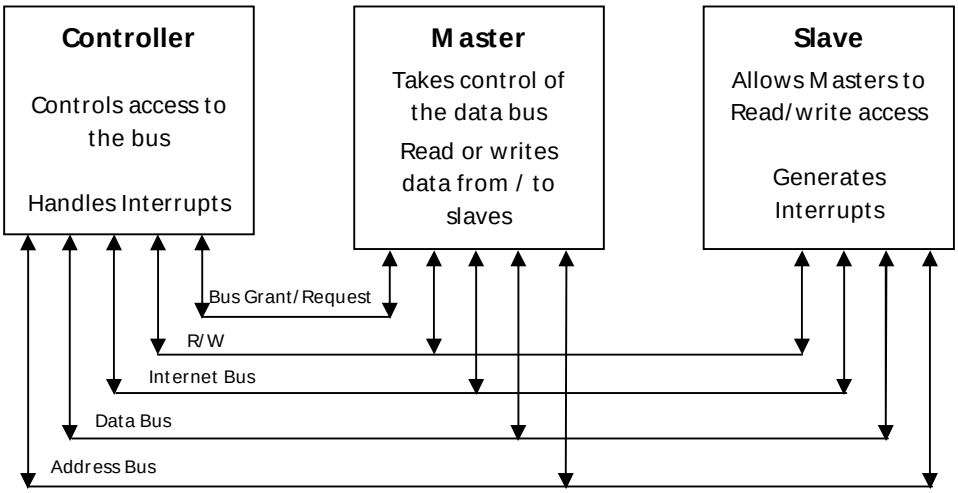


Figure 2.6: VME bus System

The backplane bus is made of signal lines and connectors. A special bus controller board is used to house the backplane control logic, such as the system clock driver, arbiter, bus timer and power driver.

Functional module: A functional module is collection of electronic circuitry that resides on one functional board and works to achieve special bus control function. These functions are:

- An arbitrator is a functional module that accepts bus request from the requester module and grant control of the DTB to one request at a time.
- A bus timer measures the time each data transfer takes on the DTB and terminates the DTB cycle if transfers take too long.
- An interrupter module generates an interrupt request and provides status /ID information when an interrupt handler module requests it.
- A location monitor is a functional module that monitors data transfer over the DTB. A power monitor watches the status of the power source and signals when power unstable.
- A system clock driver is a module that provides a clock timing signal on the utility bus. In addition, board interface logic is needed to match the signal line impedance, the propagation time and termination values between the backplane and the plug in board.

Asynchronous Data Transfer

All the operations in a digital system are synchronized by a clock that is generated by a pulse generator. The CPU and I/O interface can be designed independently or they can share common bus. If CPU and I/O interface share a common bus, the transfer of data between two units is said to synchronous. There are some disadvantages of synchronous data transfer, such as:

- It is not flexible, as all bus devices run on the same clock rate.
- Execution times are the multiples of clock cycles (if any operation needs 3.1 clock cycles, it will take 4 cycles).
- Bus frequency has to be adapted to slower devices. Thus, one cannot take full advantage of the faster ones.
- It is particularly not suitable for an I/O system in which the devices are comparatively much slower than processor.

In order to overcome all these problems, an asynchronous data transfer is used for input/ output system.

The word 'asynchronous' means not in step with the elapse of time; In case of asynchronous data transfer, the CPU and I/O interface are independent of each other. Each uses its own internal clock to control its registers. There are two popular techniques used for such data transfer: strobe control and handshaking.

Strobe Control

In strobe control**,** a control signal, called strobe pulse, which is supplied from one unit to other, indicates that data transfer has to take place. Thus, for each data transfer, a strobe is activated either by source or destination unit. A strobe is a single control line that informs the destination unit that a valid data is available on the bus. The data bus carries the binary information from source unit to destination unit.

Data transfer from source to destination

The steps involved in data transfer from source to destination are as follows: (i) The source unit places data on the data bus.

(ii) A source activates the strobe after a brief delay in order to ensure that data values are steadily placed on the data bus.

(iii) The information on data bus and strobe signal remain active for some time that is sufficient for the destination to receive it.

(iv) After this time the sources remove the data and disable the strobe pulse, indicating that data bus does not contain the valid data.

(v) Once new data is available, strobe is enabled again.

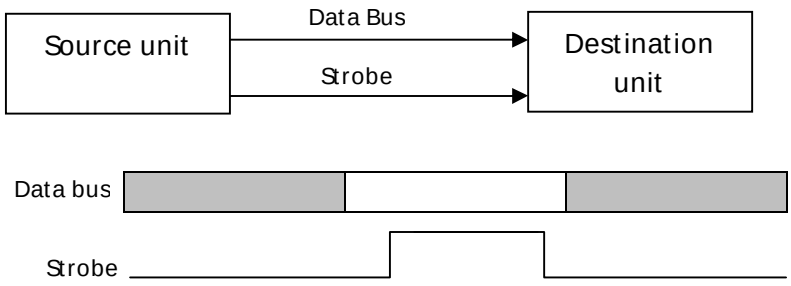


Figure 2.7: Source- Initiated Strobe for Data Transfer

Data transfer from destination to source

The steps involved in data transfer from destination to source are as follows:

1. The destination unit activates the strobe pulse informing the source to provide the data.
2. The source provides the data by placing the data on the data bus.
3. Data remains valid for some time so that the destination can receive it.
4. The falling edge of strobe triggers the destination register.
5. The destination register removes the data from the data bus and disables the strobe.

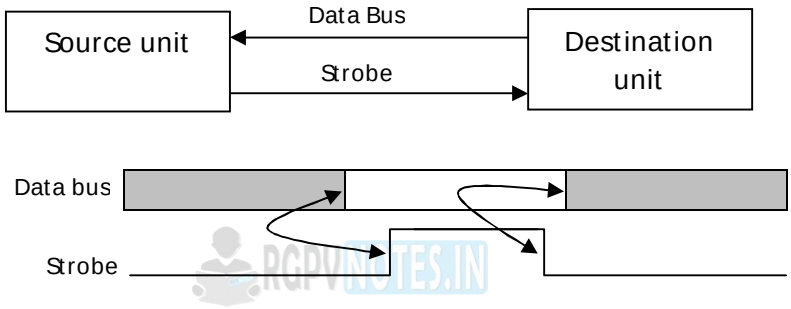


Figure 2.8: Destination- Initiated Strobe for Data Transfer

The disadvantage of this scheme is that there is no surety that destination has received the data before source removes the data. Also, destination unit initiates the transfer without knowing whether source has placed data on the data bus.

Thus, another technique, known as handshaking, is designed to overcome these drawbacks.

Handshaking

The handshaking technique has one more control signal for acknowledgement that is used for intimation. As in strobe control, in this technique also, one control line is in the same direction as data flow, telling about the validity of data. Other control line is in reverse direction telling whether destination has accepted the data.

Data transfer from source to destination

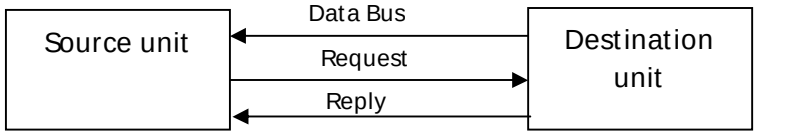
In this case, there are two control lines as shown in figure request and reply. The sequence of actions taken is as follows

(i) Source initiates the data transfer by placing the data on data bus and enable request signal.

(ii) Destination accepts the data from the bus and enables the reply signal.

(iii) As soon as source receives the reply, it disables the request signal. This also invalidates the data on the bus.

(iv) Source cannot send new data until destination disables the reply signal. (v) Once destination disables the reply signal, it is ready to accept new signal.



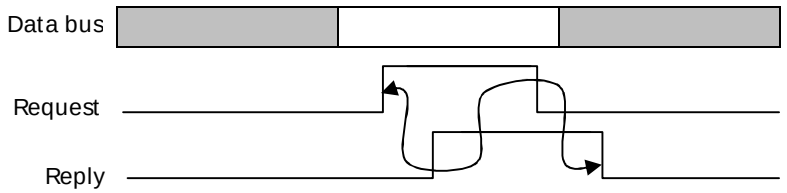


Figure 2.9: Source-Initiated Data Transfer Using Handshaking Technique

Data transfer from destination to source

The steps taken for data transfer from destination to source are as follows:

(i) Destination initiates the data transfer sending a request to source to send data telling the latter that it is ready to accept data.

(ii) Source on receiving request places data on data bus.

(iii) Also, source sends a reply to destination telling that it has placed the requisite data on the data bus and has disabled the request signal so that destination does not have new request until it has accepted the data.

(iv) After accepting the data, destination disables the reply signal so that it can issue a fresh request for data.

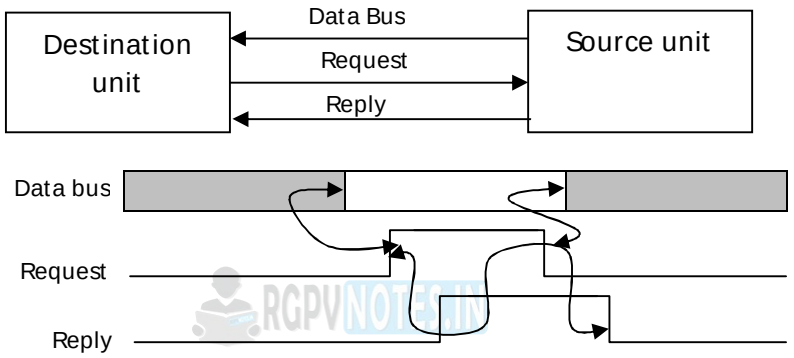


Figure 2.10: Destination-Initiated Data Transfer Using Handshaking Technique

Advantage of asynchronous bus transaction

- It is not clocked.
- It can accommodate a wide range of devices.

Arbitration transaction (Bus Arbitration)

Since at a unit time only one device can transmit over the bus, hence one important issue is to decide who should access the bus. Bus arbitration is the process of determining the bus master who has the bus control at a given time when there is a request for bus from one or more devices.

Devices connected to a bus can be of two kinds:

1. Master: is active and can initiate a bus transfer.
2. Slave: is passive and waits for requests.

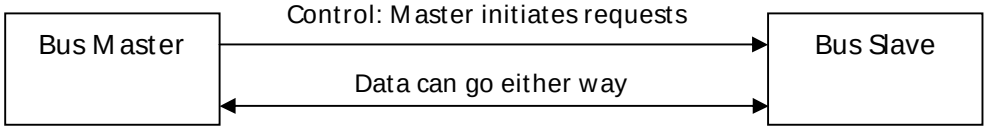


Figure 2.11: Bus arbitration

In most computers, the processor and DMA controller are bus masters whereas memory and I/O controllers are slaves. In some systems, certain intelligent I/O controllers are also bus masters. Some devices can act both as master and as slave, depending on the circumstances:

- CPU is typically a master. A coprocessor, however, can initiate a transfer of a parameter from the CPU

here CPU acts like a slave.

- An I/O device usually acts like a slave in interaction with the CPU. Several devices can perform direct access to the memory, in which case they access the bus like a master.
- The memory acts only like a slave.
- In some systems especially one where multiple processors share a bus. When more than one bus master simultaneously needs the bus, only one of them gains control of the bus and become active bus master. The others should wait for their turn. The 'bus arbiter' decides who would become current bus master. Bus arbitration schemes usually try to balance two factors:
- Bus priority: the highest priority device should be serviced first
- Fairness: Even the lowest priority device should never be completely locked out from the bus

Let's understand the sequence of events take place, where the bus arbitration consists are following:

1. Asserting a bus mastership request
2. Receiving a grant indicating that the bus is available at the end of the current cycle. A bus master cannot use the bus until its request is granted.
3. Acknowledging that mastership has been assumed.
4. A bus master must signal to the arbiter after finish using the bus.

The 68000 has three bus arbitration control pins:

BR - The bus request signal assigned by the device to the processor intending to use the buses.

BG - The bus grant signal is assigned by the processor in response to a BR, indicating that the bus will be released at the end of the current bus cycle. When BG is asserted BR can be de-asserted. BG can be routed through a bus arbitrator e.g. using daisy chain or through a specific priority-encoded circuit.

BGACK - At the end of the current bus cycle the potential bus master takes control of the system buses and asserts a bus grant acknowledge signal to inform the old bus master that it is now controlling the buses. This signal should not be asserted until the following conditions are met:

1. A bus grant has been received.
2. Address strobe is inactive, which indicates that the microprocessor is not using the bus.
3. Data transfer acknowledge is inactive, which indicates that neither memory nor peripherals are using the bus.
4. Bus grant acknowledge is inactive, which indicates that no other device is still claiming bus mastership.

On a typical I/O bus, however, there may be multiple potential masters and there is a need to arbitrate between simultaneous requests to use the bus. The arbitration can be either central or distributed.

Centralized bus arbitration in which a dedicated arbiter has the role of bus arbitration. In the central scheme, it is assumed that there is a single device (usually the CPU) that has the arbitration hardware. The central arbiter can determine priorities and can force termination of a transaction if necessary. Central arbitration is simpler and lower in cost for a uniprocessor system. It does not work as well for a symmetric multiprocessor design unless the arbiter is independent of the CPUs.

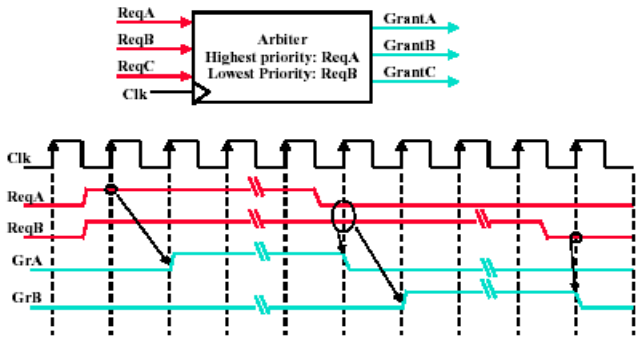


Figure 2.12: Centralized bus arbitrators

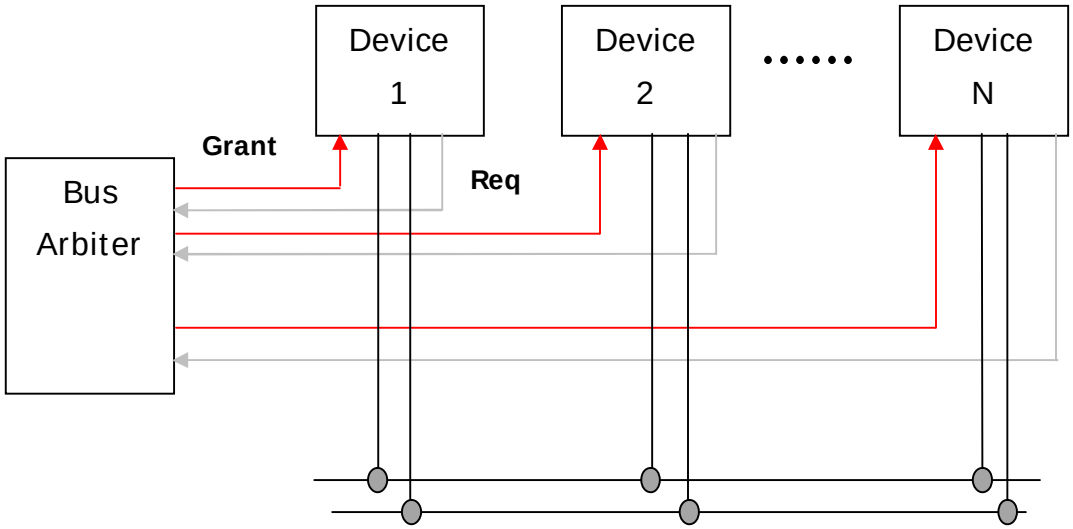


Figure 2.13: Independent requests with central arbitrator

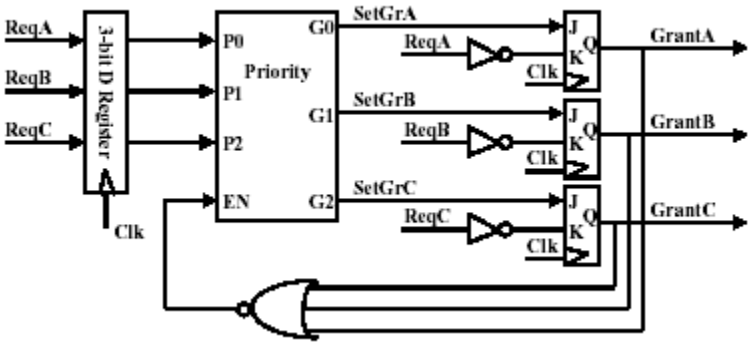


Figure 2.14: Central arbitrator

Distributed bus arbitration in which all bus masters cooperate and jointly perform the arbitration. In this case, every bus master has an arbitration section. For example: there are as many request lines on the bus as devices; each device monitors each request line after each bus cycle each device knows if he is the highest priority device which requested the busy if yes, it takes it. In the distributed scheme, every potential master carries some hardware for arbitration and all potential masters compete equally for the bus. The arbitration scheme is often based on some pre-assigned priorities for the devices, but these can be changed. Distributed arbitration can be done either by self-selection – where code indicates identity on the bus for example NuBus 16 devices, or by collision detection as an example Ethernet. NuBus has four arbitration lines. A candidate to bus master asserts its arbitration level on the 4-bit open collector arbitration bus. If a competing master sees a higher level on the bus than its own level, it ceases to compete for the bus. Each potential bus master simultaneously drives and samples the bus. When one bus master has gained bus mastership and then relinquished it, it would not attempt to re-establish bus mastership until all pending bus requests have been dealt with (fairness). **Daisy-chaining technique** is a hybrid of central and distributed arbitration. In these techniques all devices that can request are attached serially. The central arbiter issues grant signal to the closest device requesting it. Devices request the bus by passing a signal to their neighbors who are closer to the central arbiter. If a closer device also requests the bus, then the request from the more distant device is blocked i.e., the priority scheme is fixed by the device's physical position on the bus, and cannot be changed in software. Sometimes, multiple request and grant lines are used with daisy-chaining to enable requests from devices to bypass a closer device, and thereby implement a restricted software- controllable priority scheme. Daisy-chaining is low cost technique and also susceptible to faults. It is may lead to starvation for distant devices if a high priority devices (one nearest to arbitrator) frequently request for bus.

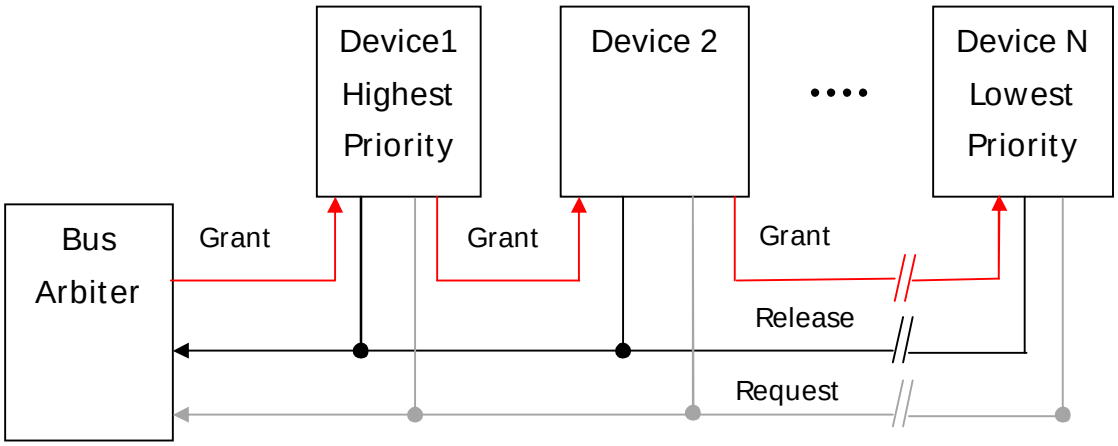


Figure 2.15: Daisy chaining arbitration scheme

Interrupt Mechanisms

A request from I/O or other peripherals to a processor is known as interrupt for service or attention. To pass the interrupt signals, a priority interrupt bus is employed. To serve as an interrupt handler, a functional module is used. Priority interrupts are handled at various levels. The interrupter must give identification details and status. The VME bus utilizes seven interrupt-request lines. To handle multiple interrupts, maximum seven interrupt handlers are used.

Using the data bus lines on a time sharing basis, interrupts can also be controlled by message passing. The saving of dedicated interrupt lines is obtained at the expense of requiring some bus cycles for handling message based interrupts. Virtual interrupts is the use of time shared data bus lines to implement interrupts.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in

RGPVNOTES.IN

Subject Name: **Advanced Computer Architecture**

Subject Code: **CS-6001**

Semester: **6th**

Subject Name: Advance Computer Architecture **Subject Code:** CS 6001

Subject Notes

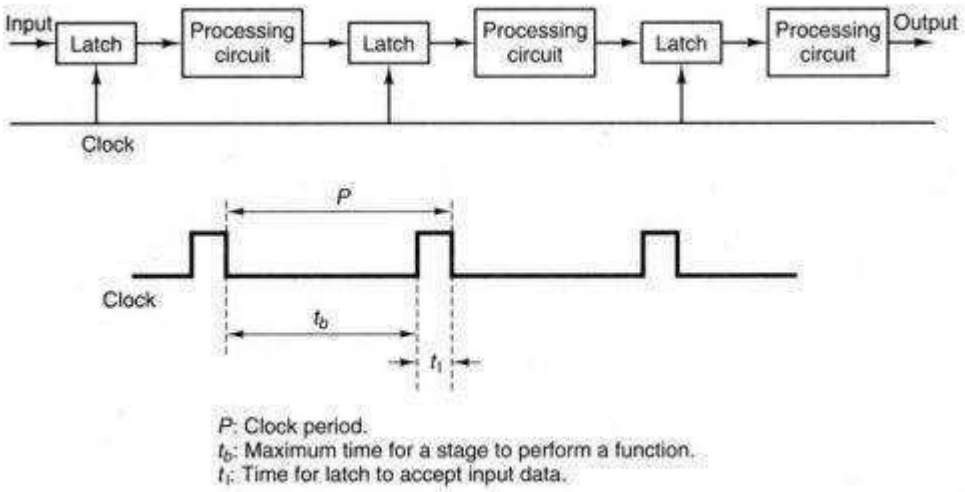
Unit-III

Pipelining:

Pipelining is one way of improving the overall processing performance of a processor. This architectural approach allows the simultaneous execution of several instructions. Pipelining is transparent to the programmer; it exploits parallelism at the instruction level by overlapping the execution process of instructions. It is analogous to an assembly line where workers perform specific task and pass the partially completed product to the next worker.

Linear Pipeline Processor:

The pipeline design technique decomposes a sequential process into several sub processes, called stages or segments. A stage performs a particular function and produces an intermediate result. It consists of an input latch, also called a register or buffer, followed by a processing circuit. (A processing circuit can be a combinational or sequential circuit.) The processing circuit of a given stage is connected to the input latch of the next stage (see Figure 3.1). A clock signal is connected to each input latch. At each clock pulse, every stage transfers its intermediate result to the input latch of the next stage. In this way, the final result is produced after the input data have passed through the entire pipeline, completing one stage per clock pulse. The period of the clock pulse should be large enough to provide sufficient time for a signal to traverse through the slowest stage, which is called the bottleneck (i.e., the stage needing the longest amount of time to complete). In addition, there should be enough time for a latch to store its input signals. If the clock's period, P, is expressed as P = tb + tl, then tb should be greater than the maximum delay of the bottleneck stage, and tl should be sufficient for storing data into a latch.



P: Clock period.
t_b: Maximum time for a stage to perform a function.
t_l: Time for latch to accept input data.

Figure 3.1 Basic structure of a pipeline.

Types of pipeline:

Pipelines can actually be divided into two classes:

a) Static or Linear Pipelines: These pipelines can perform one operation (Addition or Multiplication) at a time. The operation of a static pipeline can only be changed after the pipeline has been drained. (A pipeline is said to be drained when the last input data leave the pipeline.) For example, consider a static pipeline that is able

to perform addition and multiplication. Each time that the pipeline switches from a multiplication operation to an addition operation, it must be drained and set for the new operation. The performance of static pipelines is severely degraded when the operations change often, since this requires the pipeline to be drained and refilled each time. The output of the pipeline is produced from the last stage. For Diagram of static or linear pipeline you can refer to the previous Figure 3.1.

b) Dynamic or Non Linear Pipelines processor: A dynamic pipeline can perform more than one operation at a time. To perform a particular operation on an input data, the data must go through a certain sequence of stages. For example, Figure 3.2 shows a three-stage dynamic pipeline that performs addition and multiplication on different data at the same time. To perform multiplication, the input data must go through stages 1, 2, and 3; to perform addition, the data only need to go through stages 1 and 3. Therefore, the first stage of the addition process can be performed on an input data D1 at stage 1, while at the same time the last stage of the multiplication process is performed at stage 3 on a different input data D2. Note that the time interval between the initiation of the inputs D1 and D2 to the pipeline should be such that they do not reach stage 3 at the same time; otherwise, there is a collision. In general, in dynamic pipelines the mechanism that controls when data should be fed to the pipeline is much more complex than in static pipelines.

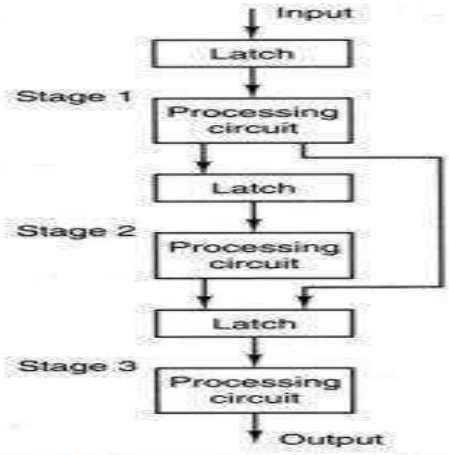


Figure 3.2 A three-stage dynamic pipeline.

Mechanism for Instruction pipeline:

In Von Neumann architecture, the process of executing an instruction involves several steps. First, the control unit of a processor fetches the instruction from the cache (or from memory). Then the control unit decodes the instruction to determine the type of operation to be performed. When the operation requires operands, the control unit also determines the address of each operand and fetches them from cache (or memory). Next, the operation is performed on the operands and, finally, the result is stored in the specified location. An instruction pipeline increases the performance of a processor by overlapping the processing of several different instructions. Often, this is done by dividing the instruction execution process into several stages. As shown in Figure 3.3, an instruction pipeline often consists of five stages, as follows:

- **Instruction fetch (IF):** Retrieval of instructions from cache (or main memory).
- **Instruction decoding (ID):** Identification of the operation to be performed.
- **Operand fetch (OF):** Decoding and retrieval of any required operands.
- **Execution (EX):** Performing the operation on the operands.
- **Write-back (WB):** Updating the destination operands.

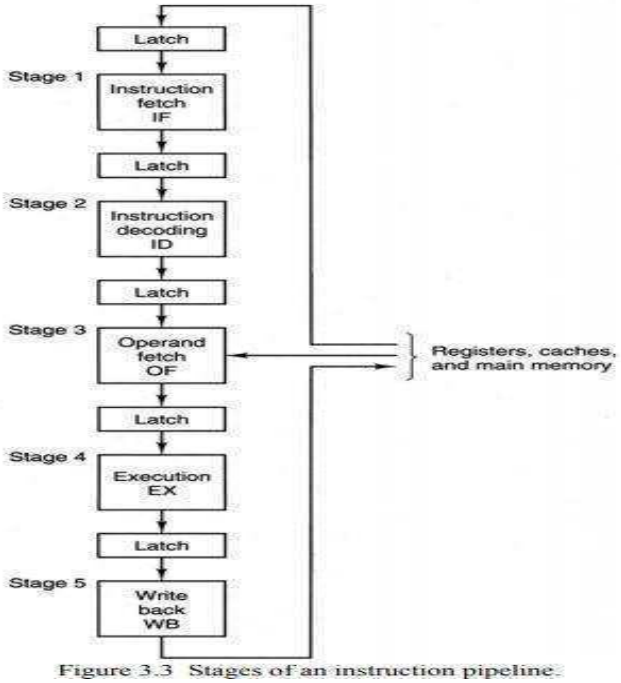


Figure 3.3 Stages of an instruction pipeline.

An instruction pipeline overlaps the process of the preceding stages for different instructions to achieve a much lower total completion time, on average, for a series of instructions. As an example, consider Figure 3.4, which shows the execution of four instructions in an instruction pipeline. During the first cycle, or clock pulse, instruction i1 is fetched from memory. Within the second cycle, instruction i1 is decoded while instruction i2 is fetched. This process continues until all the instructions are executed. The last instruction finishes the write-back stage after the eighth clock cycle. Therefore, it takes 80 nanoseconds (ns) to complete execution of all the four instructions when assuming the clock period to be 10 ns.

The total completion time can also be obtained using equation (3.1) that is,

Tpipe = m*P+(n-1)*P

=5*10+(4-1)*10

=80 ns.

Note that in a nonpipelined design the completion time will be much higher. Using equation (3.2),

Tseq = n*m*P = 4*5*10 = 200 ns.

Even though pipelining speeds up the execution of instructions, it does pose potential problems. Some of these problems and possible solutions are discussed next.

	Cycles							
	1	2	3	4	5	6	7	8
Instruction i_1	IF	ID	OF	EX	WB			
i_2		IF	ID	OF	EX	WB		
i_3			IF	ID	OF	EX	WB	
i_4				IF	ID	OF	EX	WB

Figure 3.4 Execution cycles of four consecutive instructions in an instruction pipeline.

Improving the Throughput of Instruction Pipeline:

Three sources of architectural problems may affect the throughput of an instruction pipeline. They are fetching, bottleneck, and issuing problems. Some solutions are given for each.

Fetching problem- In general, supplying instructions rapidly through a pipeline is costly in terms of chip area. Buffering the data to be sent to the pipeline is one simple way of improving the overall utilization of a pipeline. The utilization of a pipeline is defined as the percentage of time that the stages of the pipeline are used over a sufficiently long period of time. A pipeline is utilized 100% of the time when every stage is used (utilized) during each clock cycle.

Occasionally, the pipeline has to be drained and refilled, for example, whenever an interrupt or a branch occurs. The time spent refilling the pipeline can be minimized by having instructions and data loaded ahead of time into various geographically close buffers (like on-chip caches) for immediate transfer into the pipeline. If instructions and data for normal execution can be fetched before they are needed and stored in buffers, the pipeline will have a continuous source of information with which to work. Prefetch algorithms are used to make sure potentially needed instructions are available most of the time. Delays from memory access conflicts can thereby be reduced if these algorithms are used, since the time required to transfer data from main memory is far greater than the time required to transfer data from a buffer.

The bottleneck Problem

The bottleneck problem relates to the amount of load (work) assigned to a stage in the pipeline. If too much work is applied to one stage, the time taken to complete an operation at that stage can become unacceptably long. This relatively long time spent by the instruction at one stage will inevitably create a bottleneck in the pipeline system. In such a system, it is better to remove the bottleneck that is the source of congestion. One solution to this problem is to further subdivide the stage. Another solution is to build multiple copies of this stage into the pipeline.

Pipelining hazards:

If an instruction is available, but cannot be executed for some reason, a hazard exists for that instruction. These hazards create issuing problems; they prevent issuing an instruction for execution. Three types of hazard are discussed here. They are called structural hazard, data hazard, and control hazard. A structural hazard refers to a situation in which a required resource is not available (or is busy) for executing an instruction. A data hazard refers to a situation in which there exists a data dependency (operand conflict) with a prior instruction. A control hazard refers to a situation in which an instruction, such as branch, causes a change in the program flow. Each of these hazards is explained next.

Structural Hazards:

A structural hazard occurs as a result of resource conflicts between instructions. One type of structural hazard that may occur is due to the design of execution units. If an execution unit that requires more than one clock cycle (such as multiply) is not fully pipelined or is not replicated, then a sequence of instructions that uses the unit cannot be subsequently (one per clock cycle) issued for execution. A replicating and/or pipelining execution unit increases the number of instructions that can be issued simultaneously.

Another type of structural hazard that may occur is due to the design of register files. If a register file does not have multiple write (read) ports, multiple writes (reads) to (from) registers cannot be performed simultaneously.

For example, under certain situations the instruction pipeline might want to perform two register writes in a clock cycle. This may not be possible when the register file has only one write port. The effect of a structural hazard can be reduced fairly simply by implementing multiple execution units and using register files with multiple input/output ports.

Data Hazards: In a non-pipelined processor, the instructions are executed one by one, and the execution of an instruction is completed before the next instruction is started. In this way, the instructions are executed in the same order as the program. However, this may not be true in a pipelined processor, where instruction executions are overlapped. An instruction may be started and completed before the previous instruction is completed.

The data hazard, which is also referred to as the data dependency problem, comes about as a result of overlapping (or changing the order of) the execution of data-dependent instructions. For example, in Figure 3.5 instruction i2 has a data dependency on i1 because it uses the result of i1 (i.e., the contents of register R2) as input data. If the instructions were sent to a pipeline in the normal manner, i2 would be in the OF stage before i1 passed through the WB stage.

This would result in using the old contents of R2 for computing a new value for R5, leading to an invalid result. To have a valid result, i2 must not enter the OF stage until i1 has passed through the WB stage. In this way, as is shown in Figure 3.6, the execution of i2 will be delayed for two clock cycles. In other words, instruction i2 is said to be stalled for two clock cycles. Often, when an instruction is *stalled*, the instructions that are positioned after the *stalled* instruction will also be stalled. However, the instructions before the stalled instruction can continue execution. The delaying of execution can be accomplished in two ways. One way is to delay the OF or IF stages of i2 for two clock cycles.

To insert a delay, an extra hardware component called a pipeline interlock can be added to the pipeline. A pipeline interlock detects the dependency and delays the dependent instructions until the conflict is resolved. Another way is to let the compiler solve the dependency problem.

During compilation, the compiler detects the dependency between data and instructions. It then rearranges these instructions so that the dependency is not hazardous to the system. If it is not possible to rearrange the instructions, NOP (no operation) instructions are inserted to create delays.

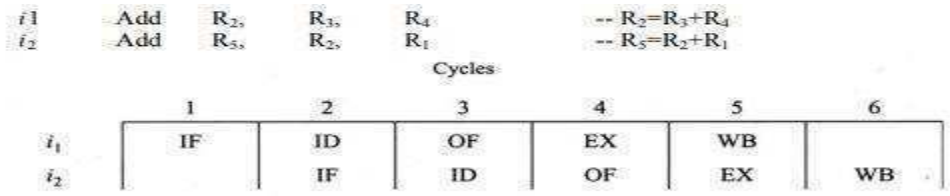


Figure 3.5 Instruction i_2 has data dependency on i_1.

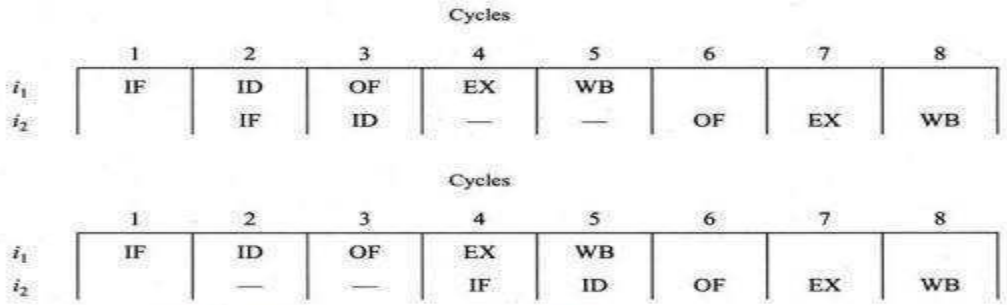


Figure 3.6 Two ways of executing data dependent instructions.

There are three primary types of data hazards named as

- RAW(Read after Write)
- WAR(Write after Read)
- WAW(Write after Write)

RAW: This type of data hazard was discussed previously; it refers to the situation in which i2 reads a data source before i1 writes to it. This may produce an invalid result since the read must be performed after the write in order to obtain a valid result.

For example, in the sequence

i1: Add R2, R3, R4 --R2=R3+R4

i2: Add R5, R2, R1 --R5=R2+R1 an invalid result may be produced if i2 reads R2 before i1 writes to it.

WAR: This refers to the situation in which i2 writes to a location before i1 reads it.

For example, in the sequence

i1: Add R2, R3, R4 --R2=R3+R4

i2: Add R4, R5, R6 --R4=R5+R6 an invalid result may be produced if i2 writes to R4 before i1 reads it; that is, the instruction i1 might use the wrong value of R4.

WAW: This refers to the situation in which i2 writes to a location before i1 writes to it.

For example, in the sequence

i1: Add R2, R3, R4 --R2=R3+R4

i2: Add R2, R5, R6 --R2=R5+R6 the value of R2 is recomputed by i2. If the order of execution were reversed, that is, i2 writes to R2.

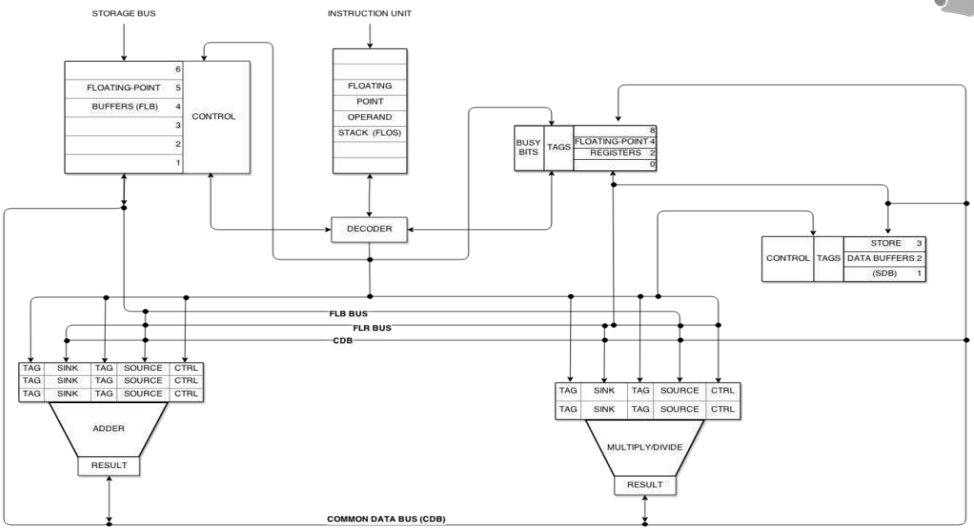
These are the methods to check the dependencies statically at the compile time by complier and by hardware at the runtime. it is not always possible to determine the actual memory addresses of load and store instructions in order to resolve a possible dependency between them. However, during the run time the actual memory addresses are known, and thereby dependencies between instructions can be determined by dynamically checking the dependency. In general, dynamic dependency checking has the advantage of being able to determine dependencies that are either impossible or hard to detect at compile time.

Here we will discuss the techniques for dynamic dependency checking. Two of the most commonly used techniques are called Tomasulo's method and the scoreboard method.

Tomasulo's Algorithm

The Tomasulo algorithm was first implemented in the IBM 360/91 Floating Point Unit which came out three years after the CDC 6600. This scheme was intedned to address several issues:

- A small number of floating point registers available the 360/91 had 4 double precision registers.
- Long memory latency this was just prior to the introduction of caches as a standard part of the memory hierarchy.
- The cost effectiveness of functional unit hardware with multiple copies of the same functional unit, some units were often underutilized.
- The performance penalties of **name dependencies**. These lead to WAW and WAR hazards.



FPU consist of:

- **Instruction buffer**
- **Load and Store buffer** : entries in this buffer consist of
  a) Busy bit: indicating the buffer element contains an outstanding load or store operation.
  b) Tag: indicating the destination (or source for store) of the data for the operation.
  c) Address (not shown) provided by the integer unit
  d) Data.
- **FP Register File** : entries in this register consist of
  a) Valid Bit: indicating the register contains the current value of the register.
  b) Tag: indicating the current source of the register value if not present.
  c) Value: the register value, if present.
- **FP functional unit with associated reservation status** :
  a) Busy bit - indicating the reservation station is occupied with an outstanding instruction.
  b) Result Tag - the "name" of the result to be produced by this instruction.
  c) Source Operands.
- **Common Data Bus (CDB**).
  Instruction executed here will occur in four phases which are fetch, issue, execute and write back.

Control Hazards (Branch handling techniques): In any set of instructions, there is normally a need for some kind of statement that allows the flow of control to be something other than sequential. Instructions that do this are included in every programming language and are called branches.

In general, about 30% of all instructions in a program are branches. This means that branch instructions in the pipeline can reduce the throughput tremendously if not handled properly. Whenever a branch is taken, the performance of the pipeline is seriously affected. Each such branch requires a new address to be loaded into the program counter, which may invalidate all the instructions that are either already in the pipeline or perfected in the buffer.

This draining and refilling of the pipeline for each branch degrade the throughput of the pipeline to that of a sequential processor. Note that the presence of a branch statement does not automatically cause the pipeline to drain and begin refilling. A branch not taken allows the continued sequential flow of uninterrupted instructions to the pipeline. Only when a branch is taken does the problem arise.

Branches can be divided into three groups:

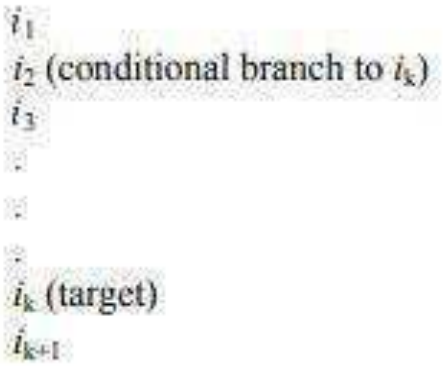
a) Unconditional Branches
b) Conditional Branches
c) Loop Branches

An unconditional branch always alters the sequential program flow. It sets a new target address in the program counter, rather than incrementing it by 1 to point to the next sequential instruction address, as is normally the case.

A conditional branch sets a new target address in the program counter only when a certain condition, usually based on a condition code, is satisfied. Otherwise, the program counter is incremented by 1 as usual. In other words, a conditional branch selects a path of instructions based on a certain condition.

If the condition is satisfied, the path starts from the target address and is called a *target path*. If it is not, the path starts from the next sequential instruction and is called a *sequential path*. Finally, a loop branch in a loop statement usually jumps back to the beginning of the loop and executes it either a fixed or a variable (data-dependent) number of times.

Among the preceding branch types, conditional branches are the hardest to handle. As an example, consider the following conditional branch instruction sequence shows the execution of this sequence in our instruction pipeline when the target path is selected. "C" denotes the numbers of cycles wasted whenever target path is chosen.



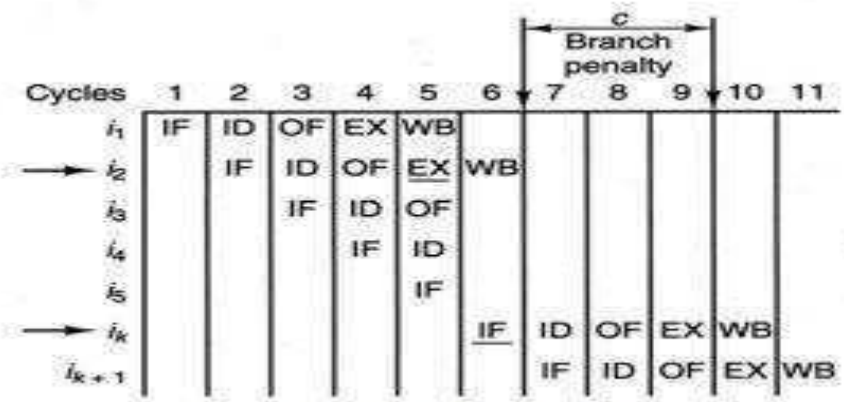


Figure 3.11 Branch Penalty.

T_{ave} = Pb * (average number of cycles per branch instruction) + (1 - Pb) * (average number of cycles per nonbranch instruction)where Pb denotes the probability that a given instruction is a branch. Thus average number of cycles per branch instruction = Pt (1+c) + (1- Pt)

Where Pt denotes the probability that branch target is chosen.

T_{ave} = Pb [Pt (1+c) + (1-Pt)(1)] + (1- Pb)(1) = 1 + cPb Pt.

Branch Prediction:

In this type of design, the outcome of a branch decision is predicted before the branch is actually executed. There are two types of predictions:

a) Static Branch Prediction: In static prediction, a fixed decision for prefetching one of the two paths is made before the program runs. For example, a simple technique would be to always assume that the branch is taken. This technique simply loads the program counter with the target address when a branch is encountered. Another such technique is to automatically choose one path (sequential or target) for some branch types and another for the rest of the branch types. If the chosen path is wrong, the pipeline is drained and instructions corresponding to the correct path are fetched; the penalty is paid.

b) Dynamic Branch Prediction: In dynamic prediction, during the execution of the program the processor makes a decision based on the past information of the previously executed branches. For example, a simple technique would be to record the history of the last two paths taken by each branch instruction. If the last two executions of a branch instruction have chosen the same path, that path will be chosen for the current execution of the branch instruction. If the two paths do not match, one of the paths will be chosen randomly.

Delayed Branching: The delayed branching scheme eliminates or significantly reduces the effect of the branch penalty.

In this type of design, a certain number of instructions after the branch instruction is fetched and executed regardless of which path will be chosen for the branch. For example, a processor with a branch delay of k executes a path containing the next k sequential instructions and then either continues on the same path or starts a new path from a new target address.

As often as possible, the compiler tries to fill the next k instruction slots after the branch with instructions that are independent from the branch instruction. NOP (no operation) instructions are placed in any remaining empty slots. As an example, consider the following code:

```
i1:  Load  R1,  A
i2:  Load  R2,  B
i3:  BrZr  R2,  i7                    -- branch to i7 if R2=0;
i4:  Load  R3,  C
i5:  Add   R4,  R2,  R3               -- R4 = R2+R3
i6:  Mul   R5,  R1,  R2               -- R5 = R1*R2
i7:  Add   R4,  R1,  R2               -- R4 = R1+R2.
```

Assume K=2, complier will modify the code by moving i_1 and inserting a NOP instruction after i_3.

```
i2:  Load  R2,  B
i3:  BrZr  R2,  i7
```

```
i1:  Load  R1,  A
     NOP
i4:  Load  R3,  C
i5:  Add   R4,  R2,  R3
i6:  Mul   R5,  R1,  R2
i7:  Add   R4,  R1,  R2.
```

Instructions after Rescheduling the i_1. And i_1 will execute regardless of the branch.

Throughput Improvement of the Instruction pipeline:

One way to increase the throughput of an instruction pipeline is to exploit instruction-level parallelism. The common approaches to accomplish such parallelism are called superscalar, superpipeline and very long instruction word (VLIW)]. Each approach attempts to initiate several instructions per cycle.

Superscalar pipeline design: The superscalar approach relies on spatial parallelism, that is, multiple operations running concurrently on separate hardware. This approach achieves the execution of multiple instructions per clock cycle by issuing several instructions to different functional units.

A superscalar processor contains one or more instruction pipelines sharing a set of functional units. It often contains functional units, such as an add unit, multiply unit, divide unit, floating-point add unit, and graphic unit. A superscalar processor contains a control mechanism to preserve the execution order of dependent instructions for ensuring a valid result.

The scoreboard method and Tomasulo's method (discussed in the previous section) can be used for implementing such mechanisms. In practice, most of the processors are based on the superscalar approach and employ a scoreboard method to ensure a valid result.

Superscalar processing has its origins in the Cray-designed CDC supercomputers, in which multiple functional units are kept busy by multiple instructions. The CDC machines could pack as many as 4 instructions in a word at once, and these were fetched together and dispatched via a pipeline. Given the technology of the time, this configuration was fast enough to keep the functional units busy without outpacing the instruction memory.

Current technology has enabled, and at the same time created the need to issue instructions in parallel. As execution pipelines have approached the limits of speed, parallel execution has been required to improve performance. As this requires greater fetch rates from memory, which hasn't accelerated comparably, it has become necessary to fetch instructions in parallel -- fetching serially and pipelining their dispatch can no longer keep multiple functional units busy. At the same time, the movement of the L1 instruction cache onto the chip has permitted designers to fetch a cache line in parallel with little cost.

In some cases superscalar machines still employ a single fetch-decode-dispatch pipe that drives all of the units. For example, the Ultra SPARC splits execution after the third stage of a unified pipeline. However, it is becoming more common to have multiple fetch- decode-dispatch pipes feeding the functional units.

The choice of approach depends on tradeoffs of the average execute time vs. the speed with which instructions can be issued. For example, if execution averages several cycles, and the number of functional units is small, then a single pipe may be able to keep the units utilized. When the number of functional units grows large and/or their execution time approaches the issue time, then multiple issue pipes may be necessary.

- Being able to fetch instructions for that many pipes at once inter-pipeline interlocking
- Reordering of instructions for multiple interlocked pipelines.
- Multiple write-back stages.
- Multiport D-cache and/or register file, and/or functionally split register file.

Reordering may be either static (compiler) or dynamic (using hardware lookahead). It can be difficult to combine the two approaches because the compiler may not be able to predict the actions of the hardware reordering mechanism.

Superscalar operation is limited by the number of independent operations that can be extracted from an instruction stream. It has been shown in early studies on simpler processor models, that this is limited, mostly by branches, to a small number (<10, typically about 4). More recent work has shown that, with speculative execution and aggressive branch prediction, higher levels may be achievable. On certain highly regular codes, the level of parallelism may be quite high (around 50). Of course, such highly regular codes are just as amenable to other forms of parallel processing that can be employed more directly, and are also the exception rather than the rule. Current thinking is that about 6-way instruction level parallelism for a typical program mix may be the natural limit, with 4-way being likely for integer codes. Potential ILP may be three times this,

but it will be very difficult to exploit even a majority of this parallelism. Nonetheless, obtaining a factor of 4 to 6 boost in performance is quite significant, especially as processor speeds approach their limits.
Going beyond a single instruction stream and allowing multiple tasks (or threads) to operate at the same time can enable greater system throughput. Because these are naturally independent at the fine-grained level, we can select instructions from different streams to fill pipeline slots that would otherwise go vacant in the case of issuing from a single thread. In turn, this makes it useful to add more functional units. We shall further explore these multithreaded architectures later in the course.

Superpipeline processor design: The superpipeline approach achieves high performance by overlapping the execution of multiple instructions on one instruction pipeline. A superpipeline processor often has an instruction pipeline with more stages than a typical instruction pipeline design.
In other words, the execution process of an instruction is broken down into even finer steps. By increasing the number of stages in the instruction pipeline, each stage has less work to do. This allows the pipeline clock rate to increase (cycle time decreases), since the clock rate depends on the delay found in the slowest stage of the pipeline. Super pipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are supported by the pipeline at a given moment. For example if we divide each stage into two, the clock cycle period t will be reduced to the half, t/2; hence, at the maximum capacity, the pipeline produces a result every t/2 s. For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance. A solution to further improve speed is the superscalar architecture.
Given a pipeline stage time T, it may be possible to execute at a higher rate by starting operations at intervals of T/n. This can be accomplished in two ways:
Further divide each of the pipeline stages into n substages.
Provide n pipelines that are overlapped.
The first approach requires faster logic and the ability to subdivide the stages into segments with uniform latency. It may also require more complex inter-stage interlocking and stall-restart logic. The second approach could be viewed in a sense as staggered superscalar operation, and has associated with it all of the same requirements except that instructions and data can be fetched with a slight offset in time. In addition, inter-pipeline interlocking is more difficult to manage because of the sub-clock period differences in timing between the pipelines. Even so, staggered clock pipelines may be necessary with superscalar designs in the future, in order to reduce peak power and corresponding power-supply induced noise. Alternatively, designs may be forced to shift to a balanced mode of circuit operation in which logic transitions are balanced by reverse transitions -- a technique used in the Cray supercomputers that resulted in the computer presenting a pure DC load to the power supply, and greatly reduced noise in the system.
Inevitably, superpipelining is limited by the speed of logic, and the frequency of unpredictable branches. Stage time cannot productively grow shorter than the interstage latch time, and so this is a limit for the number of stages.
The MIPS R4000 is sometimes called a superpipelined machine, although its 8 stages really only split the I-fetch and D-fetch stages of the pipe and add a Tag Check stage. Nonetheless, the extra stages enable it to operate with higher throughput. The UltraSPARC's 9-stage pipe definitely qualifies it as a superpipelined machine, and in fact it is a Super-Super design because of its superscalar issue. The Pentium 4 splits the pipeline into 20 stages to enable increased clock rate. The benefit of such extensive pipelining is really only gained for very regular applications such as graphics. On more irregular applications, there is little performance advantage.

Static Arithmetic Pipeline

Some functions of the arithmetic logic unit of a processor can be pipelined to maximize performance. An arithmetic pipeline is used for implementing complex arithmetic functions like floating-point addition, multiplication, and division. These functions can be decomposed into consecutive sub functions. For example

Figure presents pipeline architecture for the floating-point addition can be divided into three stages: mantissas alignment, mantissas addition, and result normalization.

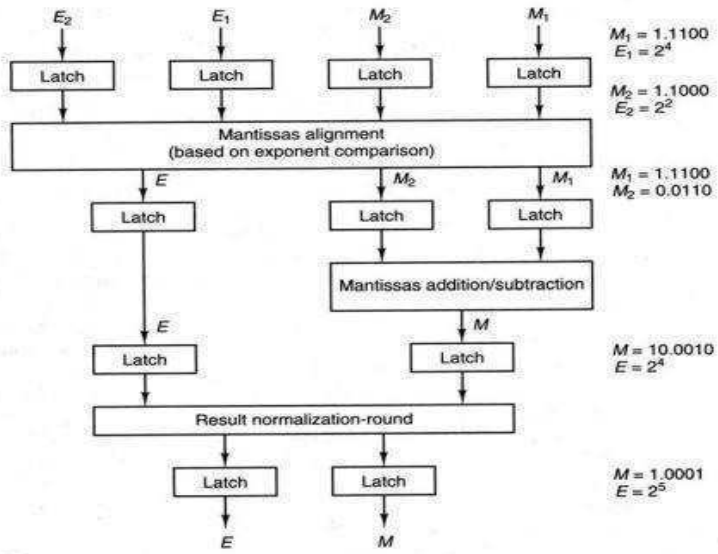


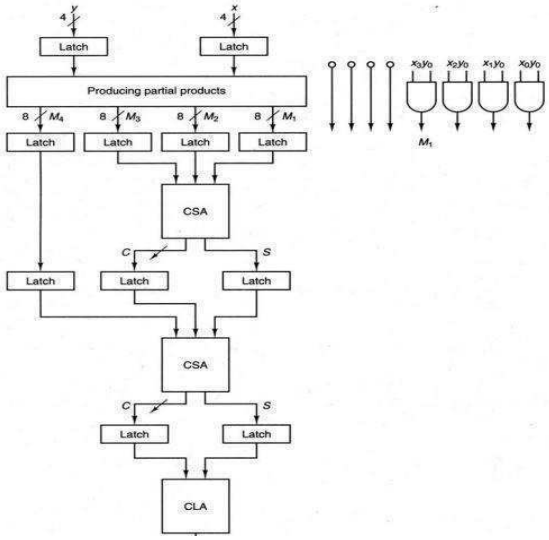
Figure 3.13 A pipelined floating-point adder.

In the first stage, the mantissas M1 and M2 are aligned based on the difference in the exponents E1 and E2. If | E1 - E2 | = k > 0, then the mantissa with the smaller exponent is right shifted by k digit positions. In the second stage, the mantissas are added (or subtracted). In the third stage, the result is normalized so that the final mantissa has a nonzero digit after the fraction point. When necessary, this normalized adjustment is done by shifting the result mantissa and the exponent.

Multifunctional arithmetic pipeline:

In arithmetic pipeline is similar to an assembly line in a factory. Data enters a stage of pipeline, which performs some arithmetic operation on data. The results are then passed to the next stage, which performs its operation and so on until the final computation has been performed.

- Each stage performs only its specific function, it does not have to be capable of performing the task of any other stage. An individual stage might be an adder or multiplier or other hardware to perform some arithmetic function.
- Variations on arithmetic pipeline are fixed arithmetic pipeline.
- It is not very useful. Unless the exact function performed by the pipeline is required, the CPU cannot use the fixed arithmetic pipeline. Configurable arithmetic pipeline.
- It is better suitable as it uses multiplexer as its input. The control unit of CPU sets the select signals of the multiplexer to control flow of data (i.e. pipeline is configurable). Vectored arithmetic unit.
- A CPU may include a vectored arithmetic unit. A vectored arithmetic unit contains multiple functional units to perform addition, multiplication, shifting, division etc) to operate different arithmetic operations in parallel.
- It is used to implement floating point operations, multiplication of fixed point numbers and similar computations encountered in scientific operations.
- Although arithmetic pipelines can perform many iterations of the same operation in parallel, they cannot perform different operations simultaneously.



This figure presents a pipelined architecture for multiplying two unsigned 4-bit numbers using carry save adders. The first stage generates the partial products M1, M2, M3, and M4. Figure represents how M1 is generated; the rest of partial products can be generated in the same way. The M1, M2, M3, and M4, are added together through the two stages of carry save adders and the final stage of carry lookahead adder.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in

RGPVNOTES.IN

Subject Name: **Advanced Computer Architecture**

Subject Code: **CS-6001**

Semester: **6th**

Subject Name: Advance Computer Architecture **Subject Code:** CS 6001

Subject Notes

Unit-IV

Cache Coherence and Synchronization

Cache coherence problem

An important problem that must be addressed in many parallel systems - any system that allows multiple processors to access (potentially) multiple copies of data - is cache coherence. The existence of multiple cached copies of data creates the possibility of inconsistency between a cached copy and the shared memory or between cached copies themselves.

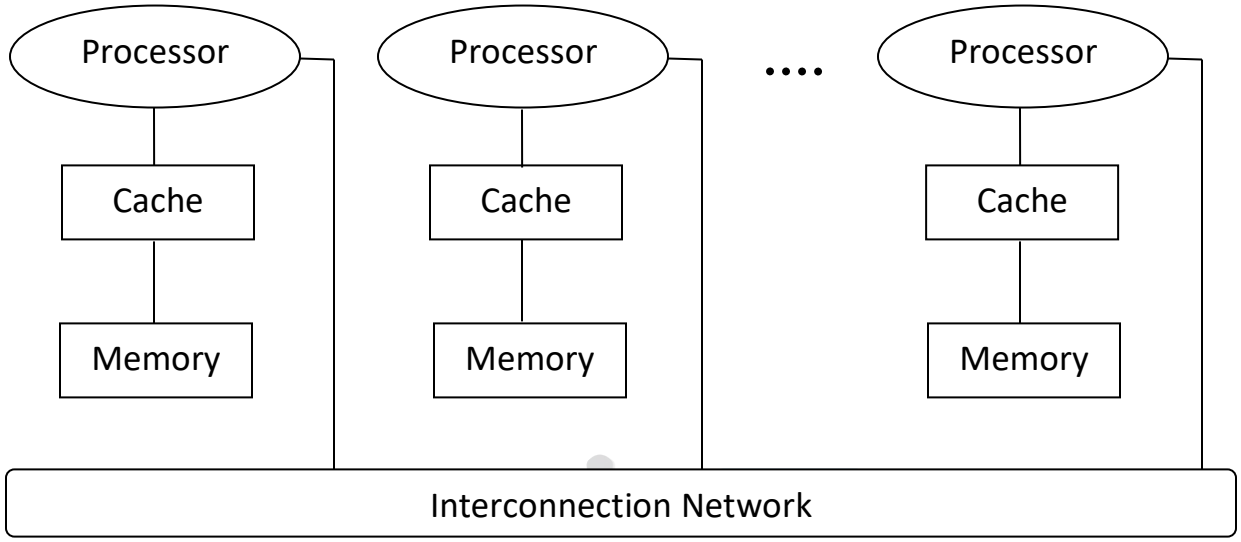


Figure 4.1: Cache coherence problem in multiprocessor

There are three common sources of cache inconsistency:

- Inconsistency in data sharing: In a memory hierarchy for a multiprocessor system data inconsistency may occur between adjacent levels or within the same level. The cache inconsistency problem occurs only when multiple private cache are used. Thus it is, the possible that a wrong data being accessed by one processor because another processor has changed it, and not all changes have yet been propagated. Suppose we have two processors, A and B, each of which is dealing with memory word X, and each of which has a cache. If processor A changes X, then the value seen by processor B in its own cache will be wrong, even if processor A also changes the value of X in main memory (which it - ultimately -should).

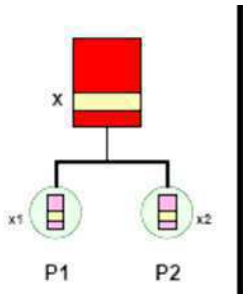


Figure 4.2: Cache coherence problem

In above example initially, x1 = x2 = X = 5. P1 writes X:=10 using write-through.
P2 now reads X and uses its local copy x2, but finds that X is still 5.

Thus P2 does not know that P1 modified X.

Thus the cache inconsistency problem occurs when multiple private cache are used and especially the problem arose by writing the shared variables.

- Process migration (even if jobs are independent): This problem occurs when a process containing shared variable X migrates from process 1 to process2 using the write back cache on the right. Thus another important aspect of coherence is serialization of writes - that is, if two processors try to write 'simultaneously', then (i) the writes happen sequentially (and it doesn't really matter who gets to write first - provided we have sensible arbitration); and (ii) all processors see the writes as occurring in the same order. That is, if processors A and B both write to X, with A writing first, then any other processors (C, D, E) all see the same thing.
- DMA I/O – this inconsistency problem occur during the I/O operation that bypass the cache. This problem is present even in a uniprocessor and can be removed by OS cache flushes)
- In practice, these issues are managed by a memory bus, which by its very nature ensures write serialization, and also allows us to broadcast invalidation signals (we essentially just put the memory address to be invalidated on the bus). We can add an extra valid bit to cache tags to mark then invalid. Typically, we would use a write-back cache, because it has much lower memory bandwidth requirements. Each processor must keep track of which cache blocks are dirty - that is, that it has written to - again by adding a bit to the cache tag. If it sees a memory access for a word in a cache block it has marked as dirty, it intervenes and provides the (updated) value. There are numerous other issues to address when considering cache coherence.

One approach to maintaining coherence is to recognize that not every location needs to be shared (and in fact most don't), and simply reserve some space for non-cacheable data such as semaphores, called a coherency domain.

Using a fixed area of memory, however, is very restrictive. Restrictions can be reduced by allowing the MMU to tag segments or pages as non-cacheable. However, that requires the OS, compiler, and programmer to be involved in specifying data that is to be coherently shared. For example, it would be necessary to distinguish between the sharing of semaphores and simple data so that the data can be cached once a processor owns its semaphore, but the semaphore itself should never be cached.

In order to remove this data inconsistency there are a number of approaches based on hardware and software techniques few are given below:

- No caches is used which is not a feasible solution
- Make shared-data non-cacheable this is the simplest software solution but produce low performance if a lot of data is shared
- software flush at strategic times: e.g., after critical sections, this is relatively simple technique but has low performance if synchronization is not frequent
- hardware cache coherence this can be achieved by making memory and caches coherent (consistent) with each other, in other words if the memory and other processors see writes then without intervention of the to software
- absolute coherence all copies of each block have same data at all times
- It is not necessary what is required is appearance of absolute coherence that is done by making temporary incoherence is OK (e.g., write-back cache)
- In general a cache coherence protocols consist of the set of possible states in local caches, the state in shared memory and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. There are basically two kinds of protocols depends on how writes is handled

Snooping Cache Protocol (for bus-based machines);

With a bus interconnection, cache coherence is usually maintained by adopting a "snoopy protocol", where each cache controller "snoops" on the transactions of the other caches and guarantees the validity of the cached data. In a (single-) multi-stage network, however, the unavailability of a system "bus" where transactions are broadcast makes snoopy protocols not useful. Directory based schemes are used in this case.

In case of snooping protocol processors perform some form of snooping - that is, keeping track of other processor's memory writes. ALL caches/memories see and react to ALL bus events. The protocol relies on global visibility of requests (ordered broadcast). This allows the processor to make state transitions for its cache-blocks.

Write Invalidate protocol

The states of a cache block copy changes with respect to read, write and replacement operations in the cache. The most common variant of snooping is a write invalidate protocol. In the example above, when processor A writes to X, it broadcasts the fact and all other processors with a copy of X in their cache mark it invalid. When another processor (B, say) tries to access X again then there will be a cache miss and either

- in the case of a write-through cache the value of X will have been updated (actually, it might not because not enough time may have elapsed for the memory write to complete - but that's another issue); or
- in the case of a write-back cache processor A must spot the read request, and substitute the correct value for X.

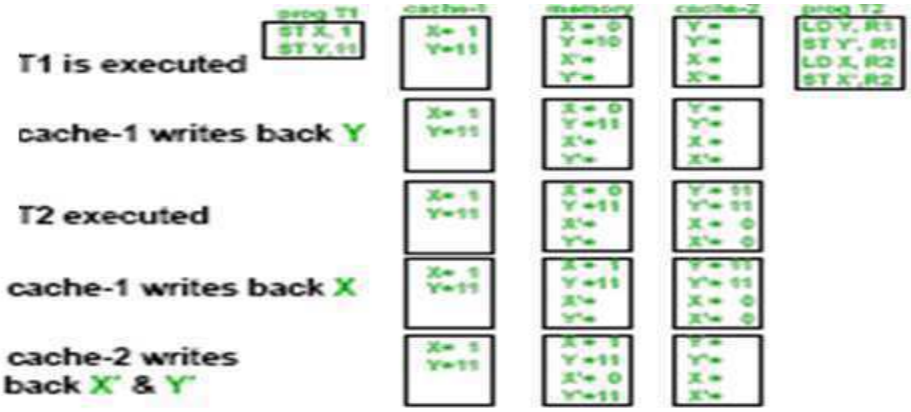


Figure 4.3: Write back with cache

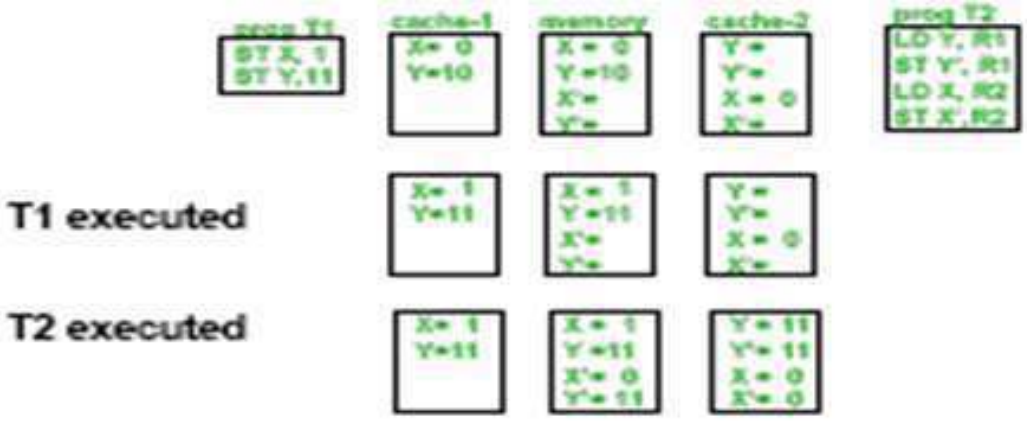


Figure 4.4: Write through with cache

An alternative (but less-common) approach is write broadcast. This is intuitively a little more obvious - when a cached value is changed, the processor that changed it broadcasts the new value to all other processors. They then update their own cached values. The trouble with this scheme is that it uses up more memory

bandwidth. A way to cut this is to observe that many memory words are not shared - that is, they will only appear in one cache. If we keep track of which words are shared and which are not, we can reduce the amount of broadcasting necessary. There are two main reasons why more memory bandwidth is used: in an invalidation scheme, only the first change to a word requires an invalidation signal to be broadcast, whereas in a write broadcast scheme all changes must be signaled; and in an invalidation scheme only the first change to any word in a cache block must be signaled, whereas in a write broadcast scheme every word that is written must be signaled. On the other hand, in a write broadcast scheme we do not end up with a cache miss when trying to access a changed word, because the cached copy will have been updated to the correct value.

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Write Broadcast for X	1	1	1
CPU B reads X		1	1	1

Figure 4.5: write back with broadcast

If different processors operate on different data items, these can be cached.

1. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic.

2. If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local.

In both cases, the coherence protocol does not add any overhead.

Directory-based Protocols

When a multistage network is used to build a large multiprocessor system, the snoopy cache protocols must be modified. Since broadcasting is very expensive in a multistage network, consistency commands are sent only to caches that keep a copy of the block. This leads to Directory Based protocols. A directory is maintained that keeps track of the sharing set of each memory block. Thus each bank of main memory can keep a directory of all caches that have copied a particular line (block). When a processor writes to a location in the block, individual messages are sent to any other caches that have copies. Thus the Directory-based protocols selectively send invalidation/update requests to only those caches having copies—the sharing set leading the network traffic limited only to essential updates. Proposed schemes differ in the latency with which memory operations are performed and the implementation cost of maintaining the directory

The memory must keep a bit-vector for each line that has one bit per processor, plus a bit to indicate ownership (in which case there is only one bit set in the processor vector).

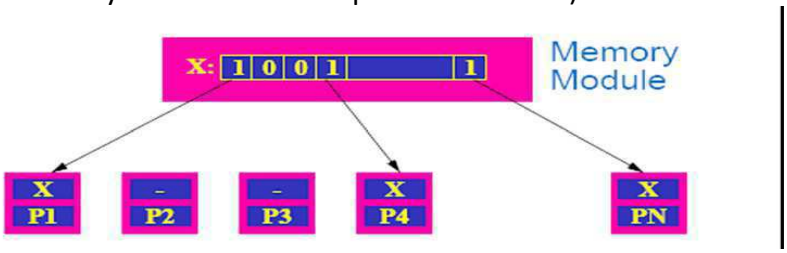


Figure 4.5: Directory based protocol

These bitmap entries are sometimes referred to as the presence bits. Only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read, write, or flush (retiring a line from cache) but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory. If

different processors operate on distinct data blocks, these blocks become dirty in the respective caches and all operations after the first one can be performed locally.

If multiple processors read (but do not update) a single data block, the data block gets replicated in the caches in the shared state and subsequent reads can happen without triggering any coherence overheads.

Various directory-based protocols differ mainly in how the directory maintains information and what information is stored. Generally speaking the directory may be central or distributed. Contention and long search times are two drawbacks in using a central directory scheme. In a distributed-directory scheme, the information about memory blocks is distributed. Each processor in the system can easily "find out" where to go for "directory information" for a particular memory block. Directory-based protocols fall under one of three categories:

Full-map directories, limited directories, and chained directories.

This full-map protocol is extremely expensive in terms of memory as it store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. It thus defeats the purpose of leaving a bus-based architecture.

A limited-map protocol stores a small number of processor ID tags with each line in main memory. The assumption here is that only a few processors share data at one time. If there is a need for more processors to share the data than there are slots provided in the directory, then broadcast is used instead.

Chained directories have the main memory store a pointer to a linked list that is itself stored in the caches. Thus, an access that invalidates other copies goes to memory and then traces a chain of pointers from cache to cache, invalidating along the chain. The actual write operation stalls until the chain has been traversed. Obviously this is a slow process.

Duplicate directories can be expensive to implement, and there is a problem with keeping them consistent when processor and bus accesses are asynchronous. For a write-through cache, consistency is not a problem because the cache has to go out to the bus anyway, precluding any other master from colliding with its access. But in a write-back cache, care must be taken to stall processor cache writes that change the directory while other masters have access to the main memory.

On the other hand, if the system includes a secondary cache that is inclusive of the primary cache, a copy of the directory already exists. Thus, the snooping logic can use the secondary cache directory to compare with the main memory access, without stalling the processor in the main cache. If a match is found, then the comparison must be passed up to the primary cache, but the number of such stalls is greatly reduced due to the filtering action of the secondary cache comparison.

A variation on this approach that is used with write-back caches is called dirty inclusion, and simply requires that when a primary cache line first becomes dirty, the secondary line is similarly marked. This saves writing through the data, and writing status bits on every write cycle, but still enables the secondary cache to be used by the snooping logic to monitor the main memory accesses. This is especially important for a read-miss, which must be passed to the primary cache to be satisfied.

The previous schemes have all relied heavily on broadcast operations, which are easy to implement on a bus. However, buses are limited in their capacity and thus other structures are required to support sharing for more than a few processors. These structures may support broadcast, but even so, broadcast-based protocols are limited.

The problem is that broadcast is an inherently limited means of communication. It implies a resource that all processors have access to, which means that either they contend to transmit, or they saturate on reception, or they have a factor of N hardware for dealing with the N potential broadcasts.

Snoopy cache protocols are not appropriate for large-scale systems because of the bandwidth consumed by the broadcast operations

In a multistage network, cache coherence is supported by using cache directories to store information on where copies of cache reside.

A cache coherence protocol that does not use broadcast must store the locations of all cached copies of each block of shared data. This list of cached locations whether centralized or distributed is called a cache directory. A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block. Distributed directory schemes

In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks. Since each memory block has an owner its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. When a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. Note that the communication overhead associated with state update messages is not reduced. Distributed directories permit O(p) simultaneous coherence operations, provided the underlying network can sustain the associated state update messages. From this point of view, distributed directories are inherently more scalable than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.

Message routing scheme

Store-and-Forward Routing packets are the basic unit of information flow in store and forward network. Each node is required to use a packet buffer and it is transmitted from the source to designation through a sequence of intermediate node. The intermediate node store the entire message in buffer before passing it on

When a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message. Suppose that a message of size m is being transmitted through such a network. Assume that it traverses l links. At each link, the message incurs a cost th for the header and twm for the rest of the message to traverse the link. Since there are l such links, the total time is (th + twm)l. Therefore, for store-and-forward routing, the total communication cost for a message of size m words to traverse l communication links is. Latency = [(message length / bandwidth) + fixed switch overhead] * #hops

In current parallel computers, the per-hop time th is quite small. For most parallel algorithms, it is less than twm even for small values of m and thus can be ignored. **Wormhole routing** in message passing was introduced in 1987 as an alternative to the traditional store-and-forward routing in order to reduce the size of the required buffers and to decrease the message latency. In wormhole routing, a packet is divided into smaller units that are called its (flow control bits) such that bits move in a pipeline fashion with the header bit of the packet leading the way to the destination node. When the header bit is blocked due to network congestion, the remaining bits are blocked as well. Switch passes message on before completely arrives and no buffering needed at switch. Latency (relative) independent of number of intermediate hops gives as below Latency = (message length / bandwidth) + (fixed switch overhead * #hops) **Asynchronous pipelining:** The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol.

Virtual channels : A virtual channel is logical link between two nodes. It is formed by a flit buffer in the source node, a physical channel between them and a flit buffer in the receiver node. Four flit buffers are used at the source node and receiver node respectively. One source buffer is paired with one receiver buffer to form a virtual channel when the physical channel is allocated for the pair. Thus the physical channel is time shared by all the virtual channels. By adding the virtual channel the channel dependence graph can be modified and one can break the deadlock cycle. Here the cycle can be converted to spiral thus avoiding a deadlock. Virtual channel can be implemented with either unidirectional channel or bidirectional channels. However a special arbitration line is needed between adjacent nodes

interconnected by bidirectional channel. This line determine the direction of information flow. The virtual channel may reduce the effective channel bandwidth available to each request. There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels.

Flow control strategies:

A routing mechanism: Determine the path a message takes through the network to get from source to destination. It takes as input a message's source and destination nodes. It may also use information about the state of the network. It returns one or more paths through the network from the source to the destination

Classification based on route selection:

A minimal routing mechanism : Always selects one of the shortest paths between the source and the destination. Each link brings a message closer to its destination but it can lead to congestion in parts of the network.

A nonminimal routing scheme: This technique may route the message along a longer path to avoid network congestion.

Classification on the basis on information regarding the state of the network:

For messaging passing scheme the for smooth control on network traffic flow that involve no congestion or deadlock situation we require to develop some strategies such that if two or more packet collide at a node competing for the buffer the policies must be set to resolve the conflict. These polices can be deterministic or adaptive routing algorithm. In deterministic routing the communication path is completely determined by a unique path for a message, based on its source and destination address. Thus here the path is uniquely predetermined in advance, independent of network condition. It does not use any information regarding the state of the network and hence may result in uneven use of the communication resources in a network.

An adaptive routing scheme : it uses information regarding the current state of the network to determine the path of the message. It detects congestion in the network and routes messages around it. Adaptive routing may depend on the network condition and alternate paths are possible. In both types of routing deadlock free algorithm is desired.

Vector Processing Principles

In computing, a vector processor or array processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items. Vector processors can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector machines appeared in the early 1970s and dominated supercomputer design through the 1970s into the 1990s, notably the various Cray platforms. The rapid fall in the price-to-performance ratio of conventional microprocessor designs led to the vector supercomputer's demise in the later 1990s.

As of 2015 most commodity CPUs implement architectures that feature instructions for a form of vector processing on multiple (vectorized) data sets, typically known as SIMD (Single Instruction, Multiple Data). Common examples include Intel x86's MMX, SSE and AVX instructions, Sparc's VIS extension, PowerPC's AltiVec and MIPS' MSA. Vector processing techniques also operate in video-game console hardware and in graphics accelerators. In 2000, IBM, Toshiba and Sony collaborated to create the Cell processor, consisting of one scalar processor and eight SIMD processors, which found use in the Sony PlayStation 3 among other applications.

Other CPU designs may include some multiple instructions for vector processing on multiple (vectorised) data sets, typically known as MIMD (Multiple Instruction, Multiple Data) and realized with VLIW (Very Long Instruction Word). Such designs are usually dedicated to a particular application and not commonly marketed for general-purpose computing. The Fujitsu FR-V VLIW/vector processor combines both technologies.

Vector Instruction types

- Vector-Vector Instructions
- Vector-Scalar Instructions
- Vector-Memory Instructions
- Vector Reduction Instructions
- Gather and Scatter Instructions
- Masking Instructions

VECTOR INSTRUCTIONS

f1: V * V
f2: V * S
f3: V x V * V
f4: V x S * V

V: Vector operand
S: Scalar operand

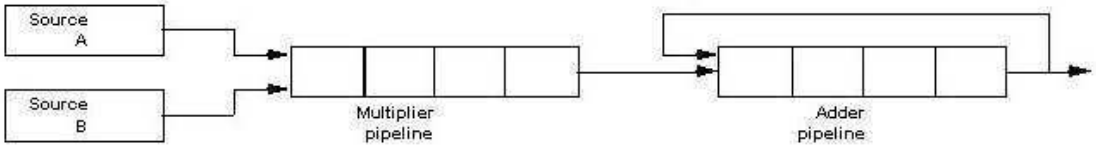
Type	Mnemonic	Description (I = 1, ..., n)	
f1	VSQR	Vector square root	B(I) * SQR(A(I))
	VSIN	Vector sine	B(I) * sin(A(I))
	VCOM	Vector complement	A(I) * $\overline{A(I)}$
f2	VSUM	Vector summation	S * Σ A(I)
	VMAX	Vector maximum	S * max{A(I)}
f3	VADD	Vector add	C(I) * A(I) + B(I)
	VMPY	Vector multiply	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 of Matrix Multiplication



$C = A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \ldots\ldots + A_k B_k$

K may be equal to 100 or even 1000

The values of A and B are either in memory or in processor registers. Each floating point adder and multiplier unit is supposed to have 4 segments. All segment registers are initially initialized to zero. Therefore the output of the adder is zero for the first 8 cycles until both the pipes are full.

Ai and Bi are brought in and multiplied at a rate of one pair per cycle. After 4 cycles the products are added to the Output of the adder. During the next 4 cycles zero is added. At the end of the 8th cycle the first four products A1B1 through A4B4 are in the four adder segments and the next four products A5 B5 through A8B8 are in the multiplier Segments.
Thus the 9th cycle and onwards starts breaking down the summation into four sections:

SIMD organization: distributed memory model and shared memory model

There are various architecture supporting parallel processing exists these are boardly classified as Multiprocessors and Multicomputers. The common classification are **Shared-Memory Multiprocessors Models which include all** UMA**:** uniform memory access (all SMP servers), NUMA: nonuniform-memory-access (Stanford DASH, SGI Origin 2000, Cray T3E) and COMA: cache-only memory architecture (KSR) Which have very low remote memory access latency.

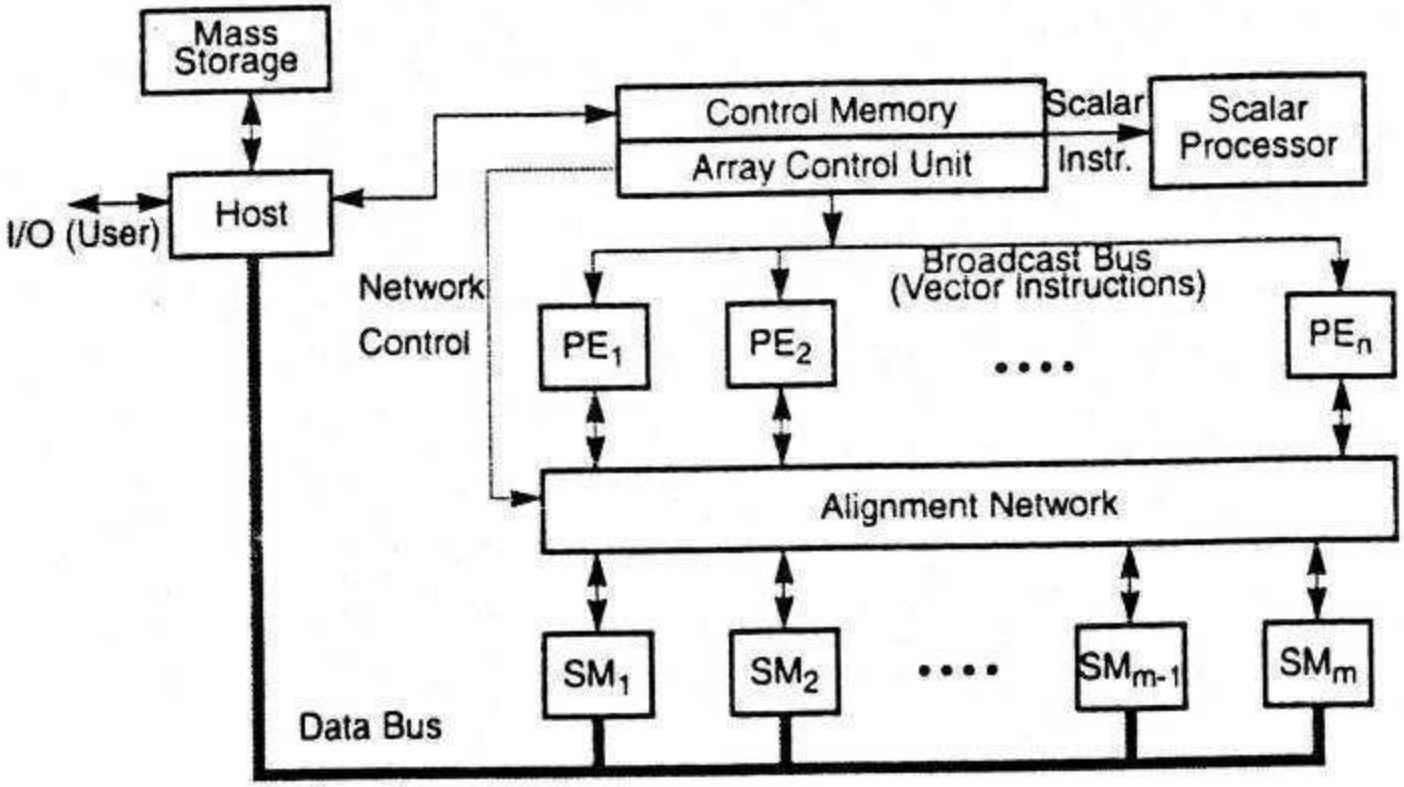


Figure 4.6: Shared memory multiprocessor

The Distributed-Memory Multicomputers Model must have a message-passing network, highly scalable like NORMA model (no-remote-memory-access), IBM SP2, Intel Paragon, TMC CM-5, INTEL ASCI Red, PC cluster

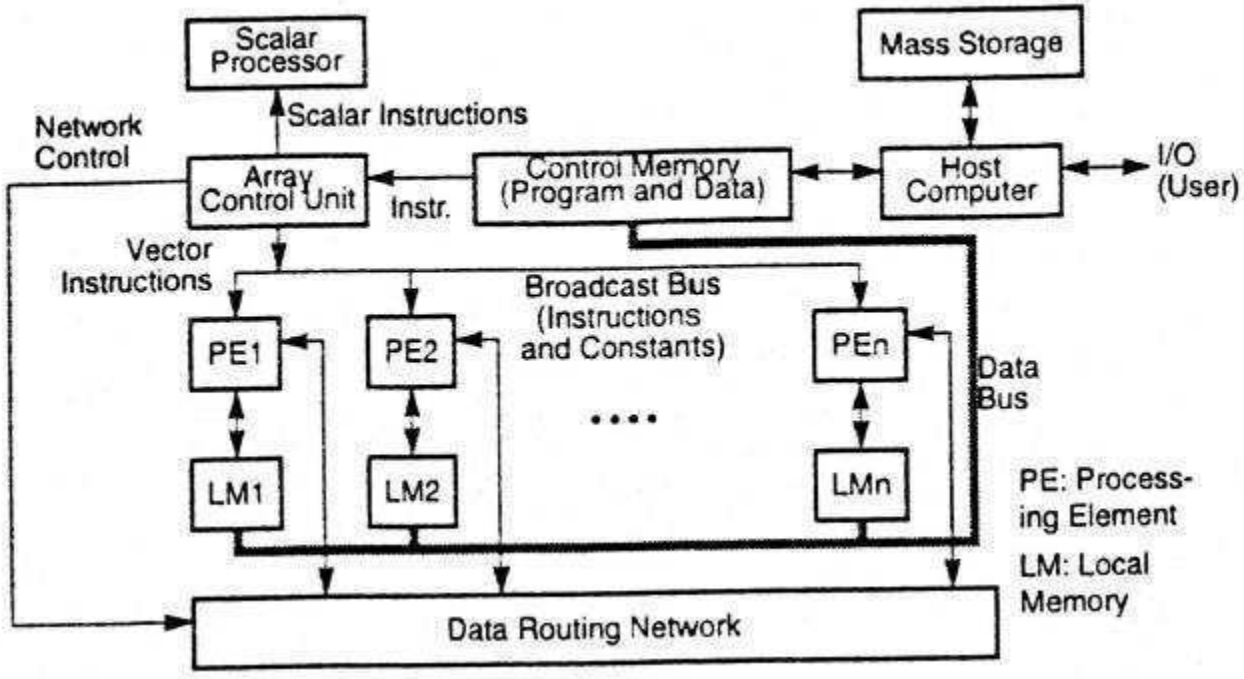


Figure 4.7: Distributed memory multiprocessor

Principles of Multithreading: Multithreading Issues and Solutions

In computer architecture, multithreading is the ability of a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system. This approach differs from multiprocessing, as with multithreading the processes and threads share the resources of a single or multiple cores: the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Where multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge from the more specialized field of transaction processing; even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains.

Two major techniques for throughput computing are multithreading and multiprocessing.

Advantages

If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

Disadvantages

Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved but can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement with its Hyper-Threading Technology, while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run in parallel. On the other hand, hand-tuned assembly language programs using MMX or AltiVec extensions and performing data prefetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.

From the software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs and operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel the software techniques used for computer multitasking. Thread scheduling is also a major problem in multithreading.

Multiple-Context Processors

Multiple context processor (mcp) architectures increase performance and reduce overhead by reducing the frequency of full context switches. In this study we accomplish this through hardware support for interprocess communication (IPC) and scheduling. Conventional scheduling techniques for single context processors do not adapt well to multiple context platforms. We present a new scheduling algorithm designed for multiple context processors which utilize information about task interaction between independent tasks (interprocess communication) to more efficiently schedule tasks on a mcp architecture, called IPC directed scheduling.

Simulation of the processor using software to simulate processor hardware was used to explore the efficacy of our design. Experimental results demonstrate the improved performance of mcp architectures over single context processors, and of IPC directed scheduling compared with conventional scheduling techniques.

Superscalar architecture is becoming the norm in today's high performance microprocessor design. However, achievable instruction level parallelism in programs limits the scalability of such architectures. In this paper, we introduce the Multiple Context Multithreaded Superscalar Processor (MCMS), which is an extension of conventional superscalar processor architecture to support multithreading. This is motivated by the enormous potential instruction level parallelism present in multithreaded programs. A hardware implementation of multithreaded constructs is also proposed. Results from trace-driven simulation show that with the MCMS, instruction level parallelism is indeed increased significantly. A MCMS processor with four hardware contexts can produce a speedup of up to 2.5 times over superscalar processor with similar hardware resources. We found that the primary limitation shifts from data dependencies in the superscalar processor to resource contentions in MCMS.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in

RGPVNOTES.IN

Subject Name: **Advanced Computer Architecture**

Subject Code: **CS-6001**

Semester: **6th**

Department of Computer Science Engineering

Subject Name: Advance Computer Architecture **Subject Code:** CS 6001

Subject Notes

Unit-V

Parallel Programming Models

The model of a parallel algorithm is developed by considering a strategy for dividing the data and processing method and applying a suitable strategy to reduce interactions. Parallel programming models are specifically designed for multiprocessor, multicomputer or vector / SIMD computers. Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed.

In this chapter, we will discuss some parallel models.

Shared-Variable Model

A program is collection of processes. Parallelism depends on how inter-process communication (IPC) is implemented. Fundamental issues in parallel programming are centered on the specification, creation, suspension, reactivation, migration, termination and synchronization of concurrent processes residing in the same or different processors.

By limiting the scope and access rights, the process address space may be shared or restricted. To ensure orderly IPC, a mutual exclusion property requires the exclusive access of a shared object by one process at a time.

Shared Variable Communication-

Multiprocessor programming is based on the use of shared variables in a common memory for IPC. As shown in figure 5.1 shared variable IPC demands the use of shared memory and mutual exclusion among multiple processes accessing the same set of variables.

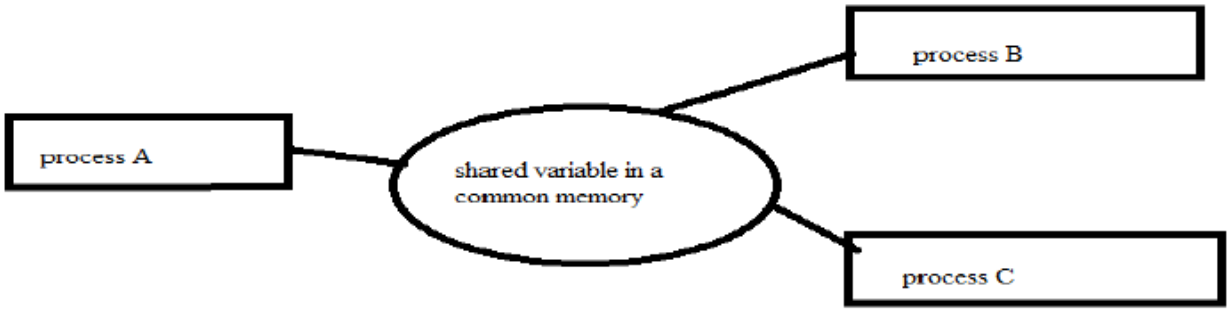


Figure 5.1: IPC using shared variable

Issues with shared variable model-

- Critical section.
- Memory consistency.
- Atomicity with memory operation.
- Fast synchronization.
- Shared data structure.

Message-Passing Model

Two processes A and B communicate with each other by passing message through a direct network. The messages may be instructions, data, synchronization, or interrupt signals, etc. Delay caused by message passing is much longer than shared variable model in a same memory. Two message passing programming models are introduced here.

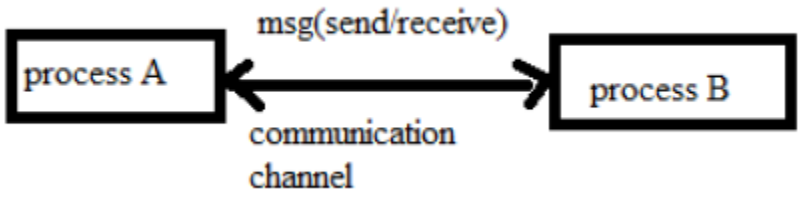


Figure 5.2: IPC using message passing

Synchronous message passing-:

- It synchronizes the sender and receiver process with time and space just like telephone call.
- No shared memory.
- No need of mutual exclusion.
- No buffer is used in communication channel.
- It can be blocked by channel being busy or error.
- One message is allowed to be transmitted via a channel at a time.
- Sender and receiver must be coupled in both time and space synchronously.
- Also called blocking communication scheme.

Asynchronous message passing-:

- Does not need to synchronize the sender and receiver in time and space.
- Non-blocking can be achieved.
- Buffers are used to hold the message along the path of connecting channel.
- Critical issue in programming this model is how to distribute or duplicate the program codes and data sets over the processing nodes.
- Message passing programming is gradually changing, once the virtual memories from all nodes are combined.

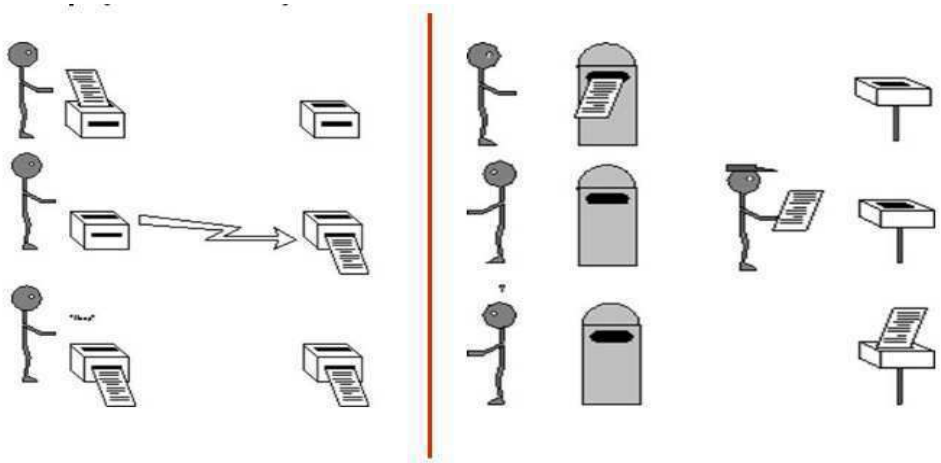


Figure 5.3: (a) Synchronous message passing (b) Asynchronous message passing

Data-Parallel Model

The data parallel code is easier to write and to debug because parallelism is explicitly handled by hardware synchronization and flow control. Data parallel languages are modified directly from standard serial programming languages.

Data parallel programming emphasizes local computations and data routing operations such as permutation, replication, reduction, and parallel prefix. It is applied to fine grain problems using regular grids, stencils, and multidimensional signal / image data sets.

In data parallel model, tasks are assigned to processes and each task performs similar types of operations on different data. Data parallelism is a consequence of single operations that is being applied on multiple data items.

Data-parallel model can be applied on shared-address spaces and message-passing paradigms. In data-parallel model, interaction overheads can be reduced by selecting a locality preserving decomposition, by using optimized collective interaction routines, or by overlapping computation and interaction.

The primary characteristic of data-parallel model problems is that the intensity of data parallelism increases with the size of the problem, which in turn makes it possible to use more processes to solve larger problems.

Example – Dense matrix multiplication

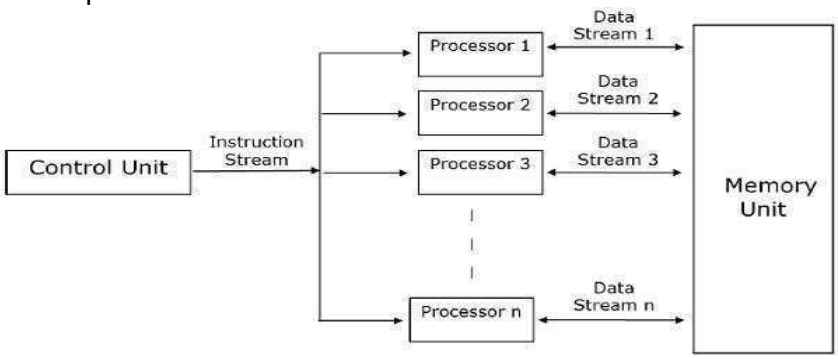


Figure 5.4: Data parallel model

Object-Oriented Model

In this model, objects are dynamically created and manipulated. Processing is performed by sending and receiving messages among objects. Concurrent programming models are built up from low level objects such as process, queues, and semaphores into high level objects like monitors and program modules.

Program abstraction leads to program modularity and software reusability as is often found in OOP. Other areas have encouraged the growth of OOP include the development of CAD tools and word processors with graphics capability. The development of concurrent object oriented programming (COOP) provides an alternative model for concurrent computing on multiprocessors or on multicomputer. Various object models differ in the internal behavior of objects and in how they interact with each other.

The idea behind parallel object-oriented programming is to provide suitable abstractions and software engineering methods for structured application design. As in the traditional object model, objects are defined as abstract data types, which encapsulate their internal state through well-designed interfaces and thus represent passive data containers. If we treat this model as a collection of shared objects, we can find an interesting resemblance with the shared data model.

The object-oriented programming model is by now well established as the state-of-the-art software engineering methodology for sequential programming, and recent developments are also emerging to establish object-orientation in the area of parallel programming. The current lack of acceptance of this model among the scientific community can be explained by the fact that computational scientists still prefer to write their programs using traditional languages. This is the main difficulty that has been faced by the object-oriented environments, though it is considered as a promising technique for parallel programming.

In short we can say in object oriented model-

- Object are created and manipulated dynamically.
- Processing is performed using object.
- Concurrent programming model are built up from low level object such as processes, queue and semaphore.

Functional and Logic Models

Two language oriented programming models for parallel processing are described in this section. The first model is based on using functional programming languages such as pure, Lisp, SISAL, and Strand 88. The second model is based on logic programming languages such as Concurrent space Prolog and Parlog.

A functional programming language emphasizes the functionality of a program and should not produce side effects after execution. There is no concept of storage, assignment, and branching in functional programs. In other words, the history of any computation performed prior to the evaluation of a functional expression should be irreverent to the meaning of the expression. The majority of parallel computers designed to support the functional model were oriented toward Lisp, such as the Multilisp developed at MIT. Other data flow computers have been used to execute functional programs, including SISAL used in the Manchester dataflow machine.

Based on predicate logic, logic programming is suitable for knowledge processing dealing with large databases. This model adopts an implicit search strategy and supports parallelism in the logic inference process. Both functional and logic programming models have been used in artificial intelligence applications where parallel processing is very much in demand.

So we can say in functional and logic models-

- Two language-oriented programming for parallel processing are purposed.
- Functional programming model such as LISP, SISAL, Strand 88.
- Logic programming model as prolog.
- Based on predicate logic, logic programming is suitable for solving large database queries.

Parallel Languages and Compilers

The environment for parallel computers is much more demanding than that for sequential computers. Software for driving parallel computers is still in the early developmental stage. Users are still forced to spend a lot of time programming hardware details instead of concentrating on program parallelism using high level abstraction. To break this hardware / software barrier, we need a parallel software environment which provides better tools for users to implement parallelism and to debug programs. Most of the recently developed software tools are still in the research and testing stage, and a few have become commercially available. A parallel language is able to express programs that are executable on more than one processor. In this we use high level language in source code as it becomes a necessity in modern computer.

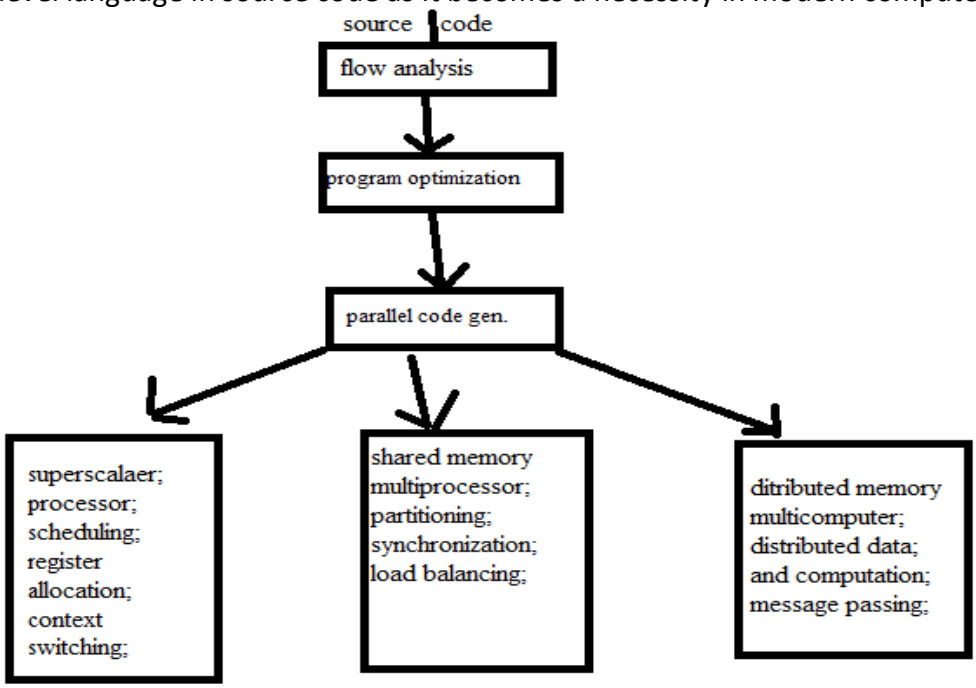


Figure 5.5: Compilation phases in parallel code generation

Role of Compiler

The role of compiler is to remove the burden of program optimization and code generation. A parallelizing compiler consists of the three major phases.

- Flow Analysis
  - Program flow pattern in order to determine data and control dependence in the source code.
  - Flow analysis is conducted at different execution levels on different parallel computers.
  - Instruction level parallelism is exploited in super scalar or VLSI processors, loop level in SIMD, vector, Task level in multiprocessors, multicomputer, or a network workstation.

- Optimization
  - The transformation of user programs in order to explore the hardware capabilities as much as possible.
  - Transformation can be conducted at the loop level, locality level, or prefetching level.
  - The ultimate goal of PO is to maximize the speed of code execution.
  - It involves minimization of code length and of memory accesses and the exploitation.
  - Sometimes should be conducted at the algorithmic level and must involve the programmer.

- Code Generation
  - Code generation usually involves transformation from one representation to another, called an intermediate form.
  - Even more demanding because parallel constructs must be included.
  - Code generation closely tied to instruction scheduling policies used.
  - Optimized to encourage a high degree of parallelism.
  - Parallel code generation is very different for different computer classes they are software and hardware scheduled.

Language Features for Parallelism

Language features are classified into six categories-

- Optimization features.
  - This features converting sequentially coded programs into parallel forms.
  - The purpose is to match the software with the hardware Parallelism in the target machine.
  - Automated parallelizer
  - Semi-automated parallelizer (programmers interaction)
  - Interactive restructure support (static analyzer, run time static, data flow graph,)

- Availability features.
  - Feature enhances user friendliness. Make the language portable to a larger class of parallel computer
  - Expand the applicability of software libraries
  - Scalability – scalable to the number of processors available and independent of hardware topology
  - Compatibility-compatible with establishment sequential
  - Portability –portable to shared memory multiprocessors, message passing multicomputer, or both

- Synchronization /communication features.
  - Single assignment languages

- Remote producer call
- Data flow languages such as ID
- Send /receive for message passing
- Barriers ,mailbox , semaphores , monitors

- Control of parallelism
  - Coarse ,medium, or fine grains
  - Explicit versus implicit parallelism
  - Global parallelism in the entire program
  - Take spilt parallelism
  - Shared task queue

- Data parallelism features
  - Used to specify how data are accessed and distributed in either SIMD and MIMD computers
  - Run- time automatic decomposition
  - Mapping specification
  - Virtual processor support
  - Direct access to shared data
  - SPMD(single program multiple data)

- Process management features
  - Needed to support the efficient creation of parallel processes, implementation of multithreading or multitasking.
  - Dynamic process creation at run time
  - Light weight processes(threads)- compare to UNIX(heavyweight)processes
  - Replicated work
  - Partitioned networks
  - Automatic load balancing

Parallel Programming Environment

An environment for parallel programming consists of hardware platforms, languages supported, OS and software tools, and application packages. The hardware platforms vary from shared memory, message passing, vector processing, and SIMD to data flow computers.

To implement a parallel algorithm you need to construct a parallel program. The environment within which parallel programs are constructed is called the parallel programming environment. There are hundreds of parallel programming environments. To understand them and organize them in a meaningful way, we need to sort them with regard to a classification scheme. Currently, various kinds of high performance machines based on the integration of many processors are available.

Key parallel programming steps:

- To find the concurrency in the given problem.
- To structure the algorithm so that concurrency can be exploited.
- To implement the algorithm in a suitable programming environment.
- To execute and tune the performance of the code on a parallel system.

Software Tools and Environments

- Tools support individual process tasks such as checking the consistency of a design, compiling a program, comparing test results, etc. Tools may be general-purpose, stand-alone tools (e.g. a word-processor) or may be grouped into workbenches.

- Workbenches support process phases or activities such as specification, design, etc. They normally consist of a set of tools with some greater or lesser degree of integration.
- Environments support all or at least a substantial part of the software process. They normally include several different integrated workbenches.

The diagram below illustrates this classification and shows some examples of these different classes of CASE support. Many types of tool and workbench have been left out of this diagram.

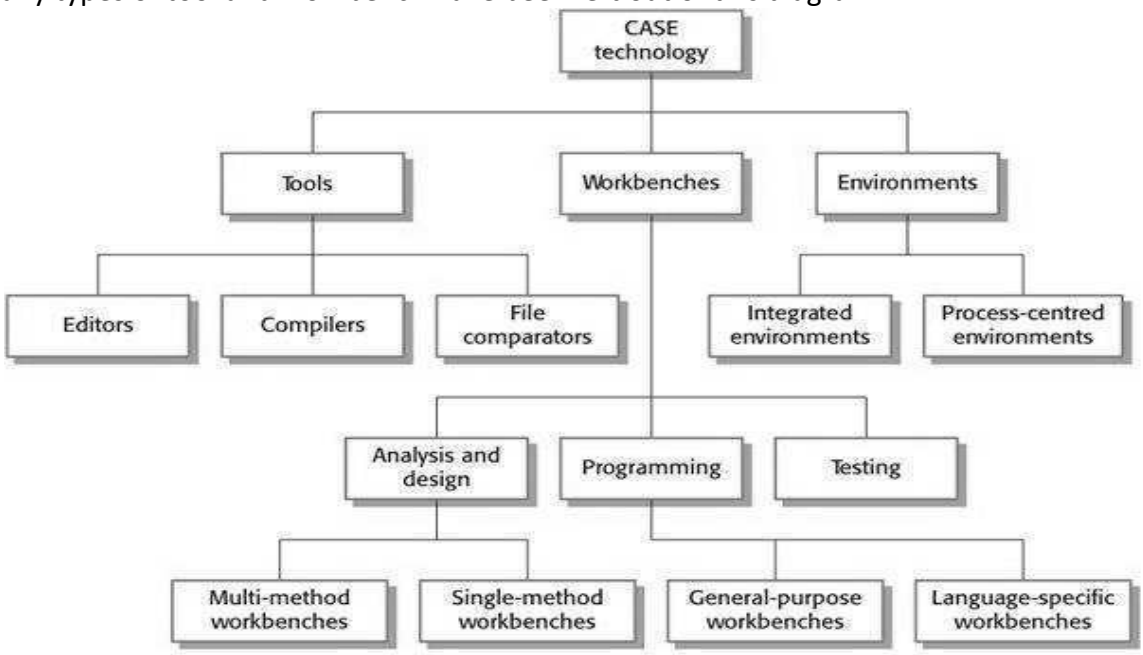


Figure 5.6: Tools, workbenches and environments

The term 'workbenches' is not now much used and the term 'environments' has been extended to cover sets of tools focused on a specific process phase e.g. programming environment or requirements engineering environment.

In the above diagram environments are classified as integrated environments or process-centered environments. Integrated environments provide infrastructure support for integrating different tools but are not concerned with how these tools are used. Process-centered environments are more general. They include software process knowledge and a process engine which uses this process model to advise engineers on what tools or workbenches to apply and when they should be used.

In practice, the boundaries between these different classes are blurred. Tools may be sold as a single product but may embed support for different activities. For example, most word processors now provide a built-in diagram editor. CASE workbenches for design usually support programming and testing so they are more akin to environments than specialized workbenches.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in