

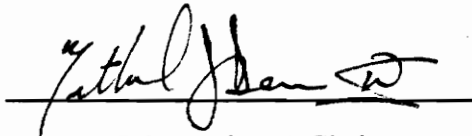
# **Simulation of Large-Scale System-Level Models**

by

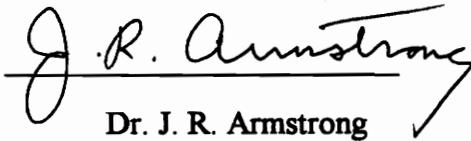
**Vikrampal Chadha**

Thesis submitted to the Faculty of the  
**Virginia Polytechnic Institute and State University**  
in partial fulfillment of the requirements for the degree of  
**Master of Science**  
in  
**Electrical Engineering**

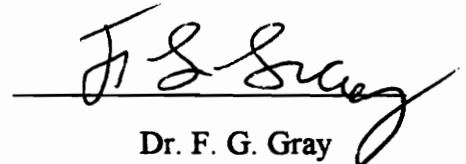
**APPROVED:**



Dr. N. J. Davis IV, Chairman



Dr. J. R. Armstrong



Dr. F. G. Gray

**September, 1994**

**Blacksburg, Virginia**

C.2

LD  
SGSS  
V8SS  
1994  
CS22  
C.2

# Simulation of Large-Scale System-Level Models

by

Vikrampal Chadha

Dr Nathaniel J. Davis IV, Chairman

Electrical Engineering

(ABSTRACT)

In this thesis, the efficient simulation of large-scale system-level VHDL models is analyzed. The system-level models chosen for the investigation are multicomputer networks, which are scalable up to thousands of processing nodes. Initially, a classification of parallel processing architectures is presented along with performance criteria and design issues related to the various interconnection network topologies. Communication and synchronization issues of MIMD systems are explored. A major limitation of planar tree structures is discussed along with a solution to help alleviate the problem, which is to make use of the binary *fat-tree*. Practical aspects of efficiently simulating large behavioral and structural models (using the fat-tree model as a case study), on a uniprocessor system are analyzed. The system resources of the workstation used to perform the simulations are carefully monitored to see where resource utilization problems usually occur. The size of the model is increased and the run time of the simulation compared with that of smaller sized models. A memory threshold level is detected after which memory resource contention problems occur and the simulation efficiency declines.

One of the problems observed in simulating complex models is the fact that simulation runs take a very long time to execute. A multicomputer using the fat-tree interconnection network is proposed as a suitable architecture for the distributed simulation of VHDL models. Various algorithms used for the parallel discrete event simulation (PDES) of VHDL models are evaluated. The feasibility of this approach is evaluated by analyzing the factors affecting the performance of the proposed architecture. The number of hops a message takes to travel from one processor to another in the fat-tree is used to estimate the time of an event message between two processors. The roll-back and communication costs amongst the processing nodes are taken into consideration when evaluating the speedup of the simulation time of a VHDL model, simulated over multiple processors. The speedup of the simulation obtained by using the fat-tree topology is compared with the results obtained with a linear array of processors. The future inclusion of the "shared variable" into the language and its impact on the implementation of parallel simulators on multicomputer networks is analyzed.

## Acknowledgments

I would like to express a deep sense of gratitude towards my advisor, Dr Nathaniel J. Davis IV, for working with me to help further my educational goals. I consider myself very fortunate to have had as a mentor, someone with the rare insight and wisdom who knew how to make me bring out the best in myself by encouraging me to fully explore those ideas that I found to be the most fascinating. In the process, I learnt a great deal and thoroughly enjoyed the research experience which led to the completion of this thesis.

Thanks are also due to my committee members, Dr J. R. Armstrong and Dr. F. G. Gray for their guidance and participation. There were numerous researchers and students around the world who spent so much of their time in helping me that I was worried that their work would suffer as a result!. I would especially like to thank Dr Peter Ashenden at the University of Adelaide, Australia, Paolo Faraboschi at the University of Genoa, Italy and Frederic Petrot at the University et Marie Curie, France for their patience in answering all my questions (and there were many!), and their quick responses at some crucial moments.

I would like to dedicate this thesis to my parents who have always been my greatest source of inspiration and who helped make this dream of mine come true.

Table of Contents

Abstract.....ii

Acknowledgements.....iii

List of Figures.....vii

List of Tables.....ix

1. Introduction.....1

    1.1 Research Goals.....2

    1.2 Outline of Thesis.....3

2. Overview of Multicomputer Networks.....5

    2.1 Introduction.....5

    2.2 Classification of Parallel Processing Architectures.....6

        2.2.1 Granularity.....8

        2.2.2 Coupling.....8

    2.3 Interconnection Topologies for Multicomputer Systems.....8

        2.3.1 Performance Criteria .....9

            2.3.1.1 Latency.....9

            2.3.1.2 Bandwidth.....9

            2.3.1.3 Cost .....9

            2.3.1.4 Reliability.....9

        2.3.2 Design Considerations.....10

            2.3.2.1 Topology.....10

            2.3.2.2 Operation Mode.....12

            2.3.2.3 Switching Strategy.....12

    2.4 Communication and Synchronization Issues.....13

        2.4.1 Synchronization Issues in Shared Memory Computers.....14

2.4.2 Synchronization Issues in Distributed Memory Computers.....	16
2.5 Scalability of Parallel Processing Systems.....	17
2.5.1 Network Scaling Factors.....	18
2.5.1.1 Node Degree.....	18
2.5.1.2 Network Diameter.....	19
2.5.1.3 Bisection Width.....	19
2.6 Summary.....	20
<b>3. Design of a Single Processing Node.....</b>	<b>21</b>
3.1 Introduction.....	21
3.2 Evolution of RISC Microprocessor Architectures.....	22
3.3 Instruction Set Architecture of the 32-bit DLX Processor.....	24
3.4 Instruction Format.....	26
3.5 Basic Steps of Execution.....	27
3.6 Structural Level VHDL Model of the Datapath.....	30
3.7 Processor Bus Architecture.....	32
3.8 Cache Design Issues.....	33
3.9 Instruction Level Parallelism.....	40
3.10 Conclusions.....	43
<b>4. Model of the Fat-tree Interconnection Network.....</b>	<b>44</b>
4.1 Introduction.....	44
4.2 Tree Terminology.....	45
4.3 Tree-based Structures.....	46
4.4 Universal Fat Trees.....	46
4.5. VHDL model of the Fat Tree.....	49
4.6 Dynamic Instruction Frequency Counts.....	52
4.7 System Resource Utilization.....	56
4.7.1. VHDL Source Code Optimization.....	57

4.7.2. Monitoring System Resources.....59

4.8 Conclusions.....63

**5. Parallel Simulation of VHDL models over a Fat-tree Interconnection Network.....65**

5.1 Introduction.....65

5.2 VHDL Parallel Discrete Event Simulation.....66

5.2.1 Sequential Discrete Event Simulation .....66

5.2.2 Centralized-queue Algorithms.....67

5.2.3 Distributed-time Algorithms.....67

5.2.3.1 Conservative Distributed-time Algorithms.....67

5.2.3.2 Optimistic Distributed-time Algorithms.....68

5.3 The VHDL Parallelism Analyzer.....68

5.4 Experimental Results and Analysis.....70

5.5 VHDL Language Extensions for System-level Modeling.....81

5.6 Conclusions.....84

**6. Conclusions.....86**

6.1 Summary of Research Work.....87

6.2 Future Work.....90

**Appendix A.....91**

**Bibliography.....93**

## List of Figures

Figure 2.1 Classification of Parallel Processing Architectures.....	8
Figure 2.2 Static interconnection topologies.....	12
Figure 2.3 Taxonomy of MIMD Computers.....	16
Figure 2.4 (a) MIMD system with distributed memory.....	16
Figure 2.4 (b) MIMD system with shared memory.....	16
Figure 3.1 An abstract view of the implementation of the processor subset.....	30
Figure 3.2 Block diagram of the 32-bit DLX processor.....	33
Figure 3.3 Processor bus read transaction.....	36
Figure 3.4 Processor bus write transaction.....	36
Figure 3.5 Block diagram of the SWIFT processor with cache and memory modules.....	39
Figure 3.6 Processor write to cache (write-through).....	40
Figure 3.7 Processor write to cache (write-back).....	41
Figure 3.8 Output result of a loop, used to add a scalar value to a vector in memory.....	44
Figure 3.9 Output result of a loop after scheduling and unrolling operations.....	44
Figure 4.1 A complete binary tree where all the switches have one parent and two children.....	49
Figure 4.2. Hop count of the messages, and message path for a 16-node fat tree.....	53
Figure 4.3 A 16-node binary fat-tree, where all non-root switches have two parents.....	55
Figure 4.4 A 16-node quadtree, where all non-root switches have two parents and four children.....	56
Figure 4.5 Comparison of the available memory resources for a 32-node fat-tree when a recursive structure is used and when a non-recursive (repetitive) structure is used.....	63



Figure 4.6 The run time is seen to increase in a non-linear way, due to memory contention,  
with an increase in the size of the fat-tree model.....63

Figure 5.1. Performance of a centralized-queue algorithm on fat-trees of different sizes.....72

Figure 5.2. Performance of a distributed-time algorithm on fat-trees of different sizes.....74

Figure 5.3. Performance of an optimistic distributed-time algorithm of a 32-node  
fat-tree with increasing rollback costs.....75

Figure 5.4 Performance of an optimistic distributed-time algorithm of a 32-node  
fat-tree with increasing communication costs.....77

Figure 5.5 Performance of an optimistic distributed-time algorithm of a 32-node  
fat-tree with increasing communication and rollback costs.....78

Figure 5.6 Performance of an optimistic distributed-time algorithm of a 32  
processor linear array with increasing communication costs.....79

Figure 5.7. Comparison of the performance decline between a 32-node fat-tree  
and a 32-node linear array with increasing communication costs.....80

## List of Tables

Table 2.1	Network Scaling Factors.....	22
Table 3.1	Comparison of the major features of some of the older CISC architectures.....	25
Table 3.2	Comparison of the major features of some of the recent RISC architectures.....	25
Table 3.3	A list of the instructions used by the Swift (DLX) processor model.....	27
Table 4.1	Dynamic instruction mix statistics of parallel benchmark programs.....	57
Table 4.2	Dynamic instruction mix statistics of benchmark programs run on a uniprocessor.....	58
Table 4.3	Dynamic instruction frequency counts obtained from the execution of a test program on the VHDL model of the 32-node fat-tree multicomputer.....	59
Table 4.4	A report on memory paging and swapping information generated using the <i>vmstat</i> command, for a 16-node fat-tree simulation on a Sun Sparksation 2.....	61
Table 4.5	A report on memory paging and swapping information generated using the <i>vmstat</i> command, for a 32-node fat-tree simulation on a Sun Sparksation 2.....	61
Table 5.1	Static and dynamic information on the VHDL models used in the tests.....	70
Table 5.2	Results obtained with Centralized and Distributed-time algorithms (rollback cost and communication cost are not considered).....	73

# **Chapter 1**

## **Introduction**

During the last decade, we have seen a major growth in the performance of low-cost computers that are based on microprocessor technology. This growth is expected to continue into the foreseeable future due to the continuing progress in increasing the density of the VLSI chips from which modern computers are constructed. In the early 1980's, microprocessor performance became a major research area with focus on quantitative metrics such as instruction execution rates and instruction counts. Sophisticated architectural techniques which had been in widespread use in large mainframes and supercomputers were incorporated into the microprocessors. The introduction of processors with a simple instruction set made it easier to migrate these techniques to the microprocessor.

During the next five years, we should see an increasing number of scalable parallel processing systems being built from these high-performance off-the-shelf microprocessors. These systems will be used to help solve the kind of single, large-scale problems (where  $N$ , the number of unknowns, is large) that overly tax the computational power of the fastest single-processor machines available today.

Multicomputers are types of parallel processing system that consist of multiple processing nodes connected by a communication network. Each processor, also referred to as a node or a processing element, has computing, memory, and communication resources. The computing resources perform the processing assigned to the node. The memory stores both program and data. The communication facilities are used to transmit and receive data from other processing nodes.

## 1.1 Research Goals

Since its introduction as a hardware description language, the VHDL language has been used only to a limited extent to model large-scale parallel processing systems. VHDL models of multicomputer networks have mainly been used to model the routing among the interconnecting switches found at the nodes of the network. The following issues are investigated in order to gain an insight into the modeling of (increasingly) complex systems with the VHDL hardware description language.

i) In order to study the suitability of using VHDL in the design of system-level models of increasing complexity, a 32-node multicomputer network based on the fat-tree architecture, with structural-level processing nodes is simulated. The processor in each computer node is tightly coupled to a memory that is physically separate and logically private from the memories of the other node computers. Interprocessor communication in the multicomputer occurs by routing messages through the interconnection network. As the size of the multicomputer model is increased from 4 to 32 processing nodes, the resources of the workstation used to perform the simulation are carefully monitored. An analysis is performed on the type of resource contention problems which occur when simulating large and complex models and the impact that these problems have on the run time of the simulation.

ii) Another aspect of the research focuses on the nature of the parallel algorithms used to program massively parallel processing systems. The von Neumann model of a computer is limited by all the communications between the central processing unit and the memory unit proceeding along what is essentially a single interconnection line, the so-called "von Neumann bottleneck." A negative aspect of this single link on the speed of a computer is not confined only to its limited capacity. In fact, the entire algorithm design process is adversely affected since we are forced to think about computational processes in sequential terms. If the dynamic instruction frequency mix of a given parallel program could be recorded from simulating the VHDL model of the multicomputer network, it would provide us with valuable information about the demands made on the processing nodes, in terms of whether the programs

are more "control" intensive or "ALU intensive." The information obtained would allow designers of future generations of microprocessors to make enhancements to better accommodate the needs of the processors used in these massively parallel processing systems.

iii) One of the problems associated with the discrete event simulation of large models is the fact that simulation runs take a very long time to execute. Parallel discrete event simulation (PDES) techniques can be used to simulate large VHDL models on parallel processing systems. Experiments are performed to evaluate the feasibility of using the fat-tree architecture to perform the parallel simulation of VHDL models and determine if a speedup in simulation time would be obtained over simulating the models on a single processor. In the case of optimistic distributed time algorithms, communication latency issues and the cost of message *rollback* are also included in the analysis in order to determine the effect of these factors on the performance of the fat-tree multicomputer network.

## 1.2 Outline of Thesis

In Chapter 2, an overview of multicomputer networks is presented. A classification of parallel processing architectures is provided along with performance criteria and design issues related to the various interconnection network topologies. Communication and synchronization issues of MIMD systems are analyzed. The factors affecting the scalability of parallel processing systems are then discussed. The chapter concludes with an overview of factors used to determine how performance characteristics scale with the number of processors being interconnected, using a specific network topology.

Chapter 3 describes the VHDL model of the processing node used in the multicomputer network. The structural-level VHDL model is based on the DLX processor, a 32-bit reduced instruction set processor described in [1]. The reasons for choosing this processor are discussed. The fact that parallel

algorithms make efficient use of the memory hierarchy is considered in the design, by providing a cache memory in each processing node. Instruction level parallelism is explored with a suitable test case.

Chapter 4 begins with a series of definitions used to describe trees in graph theory. Examples are provided of earlier research projects which utilized tree-based structures. A major limitation of the planar tree structures is analyzed, along with a solution which is to make use of the *binary fat-tree* to help alleviate the problem. The inclusion of structural-level processing nodes in the model of the fat-tree multicomputer, described in Chapter 4, has a useful consequence. The dynamic instruction frequency mix of a given parallel program can be measured.

A practical aspect of efficiently simulating large behavioral and structural models, (using the fat-tree model as a case study), on a uniprocessor system is analyzed. The system resources of the workstation used to perform the simulations are carefully monitored to see where resource utilization problems usually occur. The size of the model is increased and the run time of the simulation compared with that of smaller sized models. A memory threshold level is detected after which memory resource contention problems occur, and the simulation efficiency declines.

One of the problems observed in simulating complex models is that simulation runs take a very long amount of time to execute. The trend is for run times to lengthen, since the combinatorial complexity of circuits is out-stripping performance improvements in computers. In Chapter 5, a multicomputer using the fat-tree interconnection network is evaluated as a suitable architecture for the parallel simulation of VHDL models. Various algorithms used for the parallel discrete event simulation (PDES) of VHDL models are explored. The feasibility of this approach is carried out by analyzing the factors affecting the performance of the proposed architecture. Results obtained from simulating the model of the fat-tree multicomputer network, described in Chapter 4, are used to provide details about the interconnecting links between the various processing nodes. The number of hops a message takes to travel from one processor to another in the fat-tree is used to estimate the time of an event message between two processors. The *roll-back* cost and the cost of communication amongst the processing nodes are taken into consideration

when evaluating the *speedup* of the simulation time of a VHDL model, simulated over multiple processors. The speedup of the simulation of a VHDL model using the fat-tree topology is compared with the results obtained with a linear array topology.

The future inclusion of the "shared variable" into the language and its impact on the implementation of parallel simulators on multicomputer networks is analyzed. Conclusions drawn from the results obtained in the previous chapters are presented in Chapter 6.

## **Chapter 2**

### **Overview of Multicomputer Networks**

#### **2.1 Introduction**

During the last decade, we have seen a major growth in the performance of low-cost computers that are based on microprocessor technology. This growth is expected to continue into the foreseeable future due to the continuing progress in increasing the density of the VLSI chips from which modern computers are constructed.

During the next five years, we should see an increasing number of parallel processing systems built from these high-performance off-the-shelf microprocessors. These systems will be used to help solve the kind of single, large-scale problems (where  $N$ , the number of unknowns, is large) that overly tax the computational power of the fastest single-processor machines available today. Examples of computationally intensive problems include scientific simulation modeling, advanced computer-aided design, and real-time image processing of large-scale database and information retrieval operations [2].

In this chapter, an overview of multicomputer networks is presented. A classification of parallel processing architectures is provided along with performance criteria and design issues related to the various interconnection network topologies. Communication and synchronization issues of MIMD systems are analyzed. The factors affecting the scalability of parallel processing systems are then discussed. The chapter concludes with an overview of the factors used to determine how performance characteristics scale with the number of processors being interconnected, using a specific network topology.



## 2.2 Classification of Parallel Processing Architectures

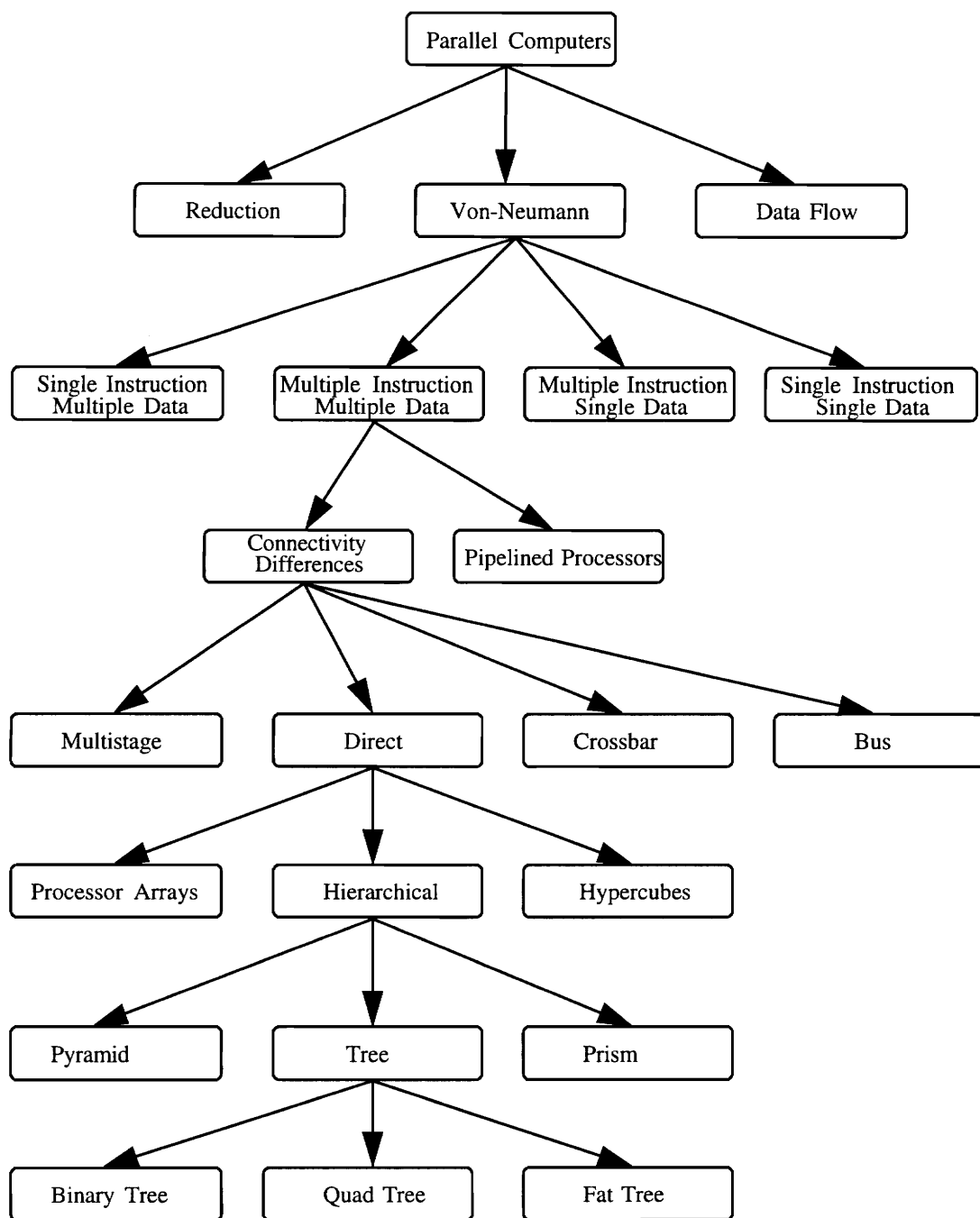
The most widely used scheme to classify the different type of processing systems was developed by Flynn [3]. It provides a four-way classification, in terms of whether one or several processors execute instructions on one or several streams of data. This scheme, though simplistic in nature, has endured the passage of time, probably because our understanding of multicomputers is still too primitive to replace it with a more definitive one.

In the case of one or more processors operating on multiple streams of data, Flynn's classification is given below:

**SIMD (Single Instruction Multiple Data Stream)** -- A network with only one controller although each processor works with information stored in its own and nearby memories. An example of such a system is the Connection Machine (CM-2) built by Thinking Machines Corporation which consists of a bit-slice array of up to 65,536 processing elements [2].

**MIMD (Multiple Instruction Multiple Data Stream)** -- A network where each processor has its own controller and can work in a completely asynchronous way, allowing multiple threads to be executed in an autonomous manner. Intel's mesh-connected Paragon system uses this model to connect its processing nodes [2]. Figure 2.1 shows a hierarchical classification of parallel processing architectures, with the MIMD model divided into a number of subclasses. Data parallelism refers to the situation where the same operation executes over a large array of data. Dataflow computers emphasize a high degree of parallelism at the fine-grain instruction level. Reduction computers are based on a demand driven mechanism which initiates an operation based on the demand for its result by other computations.

A mix of SIMD and MIMD leads to the SPMD (Single Program Multiple Data) model in which each processor executes the same SIMD program on its local data, but with an individual flow of control. In this way, the computation may be switched between SIMD lock step and MIMD asynchronous operation, based on the structure of the SPMD program. Generally, SPMD is regarded to be more of a MIMD programming style (SIMD programming on an MIMD machine) rather than a distinct classification.



**Figure 2.1** Classification of Parallel Processing Architectures

There are a number of other important distinctions to be considered which could be used to categorize parallel processing systems [4]. Among them are:

2.2.1 Granularity -- The amount of processing and communication in the system needs to be balanced in order to obtain optimal performance. An important factor that helps determine the optimal topology is the way in which the application is partitioned into individual processing tasks. The *granularity* or *grain size* is the average size of each processor's subtask, measured in number of instructions executed in a program segment. The grain size of the partition is the metric used to define the size of each processing task. In processing data, for example, processing could be done at the bit level, vector and matrix level, record level, file level and so on. A relatively small number of powerful processors would generally be used for coarse-grain parallelism with information rarely being passed from one processor to the other. At the other extreme we find fine-grain parallelism where a large number of less powerful processors are used to execute programs with a large amount of communication between the processors.

2.2.2 Coupling -- The degree of coupling between the processing elements is another method commonly used to classify parallel processing systems. A system is considered to be tightly coupled if the processors share a common memory, as in the case of multiprocessors, or if the communication between the processors is good (irrespective of the underlying hardware/software mechanisms), as in the case of multicomputers. A loosely coupled system would have its processing elements located at greater distances from one another, as can be seen with a workstation cluster (used for distributed computing).

The issue of whether the system has been designed to be used for general purpose computing or special purpose applications (digital signal processing, for example) and the different amounts and types of synchronization amongst processors are other factors which help in the overall classification.

## **2.3 Interconnection Topologies for Multicomputer Systems**

Multicomputers are types of parallel processing systems that consist of multiple processing nodes connected by a communication network. Each processor, also referred to as a node or a processing element, has computing, memory and communication resources. The computing resources perform the processing assigned to the node. The memory stores both program and data. The communication facilities are used to transmit and receive data from other processing nodes.

### **2.3.1 Performance Criteria**

The issues related to the performance of an interconnection network connecting a set of processing nodes are the following [5]:

**2.3.1.1 Latency** -- The message latency is the amount of time it takes a single message to travel between two processors. This performance metric depends on the time it takes a processor to prepare the message which is to be transmitted, the distance that the message has to travel, the amount of traffic prevalent in the network at that time and the length of time taken by the receiving node to process the message.

**2.3.1.2 Bandwidth** -- The bandwidth helps indicate how much traffic a network can handle. It is defined as the mean number of active memory modules in one transfer cycle of the interconnection network [6]. The message locality should be kept as high as possible to help preserve the available bandwidth and reduce the possibility of congestion due to too much message traffic in the network at any particular time.

**2.3.1.3 Cost** -- There are a number of factors which affect the cost of a network. The number of overall links (wires) in the network is a factor to be measured as each physical connection adds to the overall cost. The number of links required per node is also a cost factor. Additionally, the physical layout should also be as efficient as possible as packaging complexity adds to the cost. If the routers are locally controlled

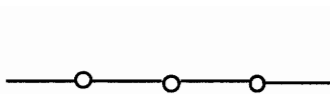
and are of a fixed degree, the cost can be minimized. Also, if the network can be built from easily available components, the cost can be greatly reduced.

**2.3.1.4 Reliability** -- The reliability of a network is an important factor which has to be carefully considered in the case of interconnection networks, where the elements may number in the thousands and the possibility of component failure is quite high. The fault-tolerance of the network can be improved by providing multiple paths to connect processing nodes to each other. In the event of a failure in any link of the network, another path could be used and the network could continue to function in a fault-tolerant manner.

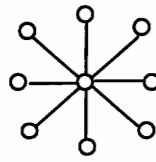
### **2.3.2 Design Considerations**

The processors and memory of a multicomputer's processing nodes are essentially the same as those that are used in single processor machines. The distinguishing feature in the case of the multicomputer processing nodes is the support for interprocessor communication that is used to enable the sharing of data between the nodes through the message-passing mechanism. Thus, the overall performance of the system is heavily dependent on the interconnection networks performance. The topology of an interconnection network defines the placement and the number of the communication links which are used to join the processing nodes in a multicomputer system. Figure 2.2 shows some of the popular *static* interconnection topologies in use today. The designer of an interconnection network has to consider a number of design issues and make a trade-off between these issues before selecting a particular topology [7]. The issues involved in making the decision are as follows:

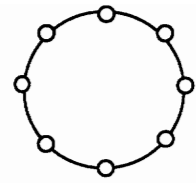
**2.3.2.1 Topology** -- The interconnection network can be broadly categorized as being static or dynamic, with each class being further broken down into their own subclasses.



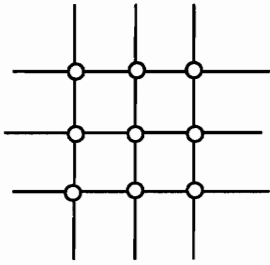
(a) Linear Array



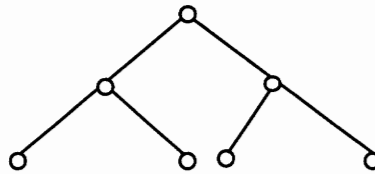
(b) Star



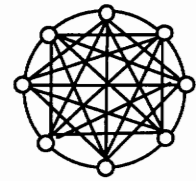
(c) Ring



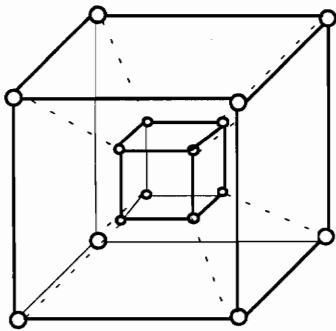
(d) Mesh



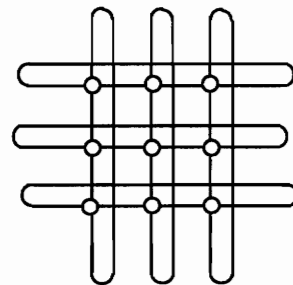
(e) Binary Tree



(f) Completely Connected



(g) Four-dimensional Hypercube



(h) Torus

**Figure 2.2** Static interconnection topologies

*Static topologies* are formed from point-to-point direct connections which will not change during program execution. They are used for fixed connections among subsystems of a centralized system or multiple computing nodes of a distributed system. They have been effectively used for problems with predictable communication patterns. Static topologies can be further classified based on the number of dimensions required for their layout. An example of a one-dimensional topology is the linear array. Two-dimensional topologies include the ring, star, tree and mesh. Three-dimensional topologies include the completely connected chordal ring and 3-cube-connected-cycles.

*Dynamic topologies* are implemented with switched channels, which are dynamically configured to match the communication demand in user programs. They have been found to be suitable for a wider range of problems than static topologies, but at a higher cost. They include buses, crossbar switches and multistage networks, which are often used in shared memory multiprocessors. Dynamic topologies are classified based on the number of stages that exist in the communication links between the nodes. A single-stage network has each communication link connected only to processing nodes. Processing nodes that do not have a direct link between them have to pass data through intermediate nodes when they want to communicate with each other. The individual links in this communication path are directly connected to pairs of processing nodes and hence these networks are also called direct networks. A multistage network consists of more than one stage of switching elements and is usually capable of connecting an arbitrary input terminal to an arbitrary output terminal.

2.3.2.2 Operation mode -- Networks can be classified as being synchronous or asynchronous, based on their mode of operation. *Synchronous* communication is used for processing in cases where communication paths are established synchronously for data manipulation functions or instruction broadcasts. The regularity of the data enables the same operations to be applied in parallel by all the processing elements, constraining hardware processes to perform in "lock-step." *Asynchronous* communication is needed for multiple processors in which communication is performed dynamically. The

individual processing elements can compete for common resources. Furthermore, the processing elements can communicate and cooperate with each other in order to improve the utilization of available resources.

**2.3.2.3 Switching Strategy** -- There are three techniques commonly used to transmit messages between processing nodes. The message can be transmitted as a whole, as in *circuit switching*, where an exclusive independent virtual circuit is established for a source/destination pair. This is accomplished by a signaling message. The path once established, is unavailable to any other source/destination pair until both the original source and destination agree to stop communicating with each other. After a path is set up, no further signaling for addressing purposes is required. One of the drawbacks of circuit switching is the fact that no communication is possible between any other source/destination pair if that pair needs to use any of the lines which are currently being used by an already established circuit.

In *packet switching*, the message is broken down into a series of small packets and then each packet is transmitted through the network from the source to the destination. The basic idea is to improve channel utilization by freeing up the channels on a path during periods in which the source and destination nodes are not communicating with one another. In the store-and-forward method, packets are routed towards their destination node without establishing a path beforehand. The packets are passed through a series of intermediate nodes. Each intermediate node receives the packets in a buffer (or store facility) where the node examines the packet's header information to determine where to forward the packet so that it gets closer to its final destination. The time it takes to receive and examine the packet headers and the time the message has to wait before a proper communication link becomes available are the main factors which affect the transmission time of a message.

Another method used to switch messages is *wormhole routing* (or cut-through switching) [8]. This method tries to combine the positive features of both circuit switching, as well as packet switching. The first packet of the message contains status information such as the source, destination and the message length. An intermediate node which receives this first packet examines it to determine where it is headed.

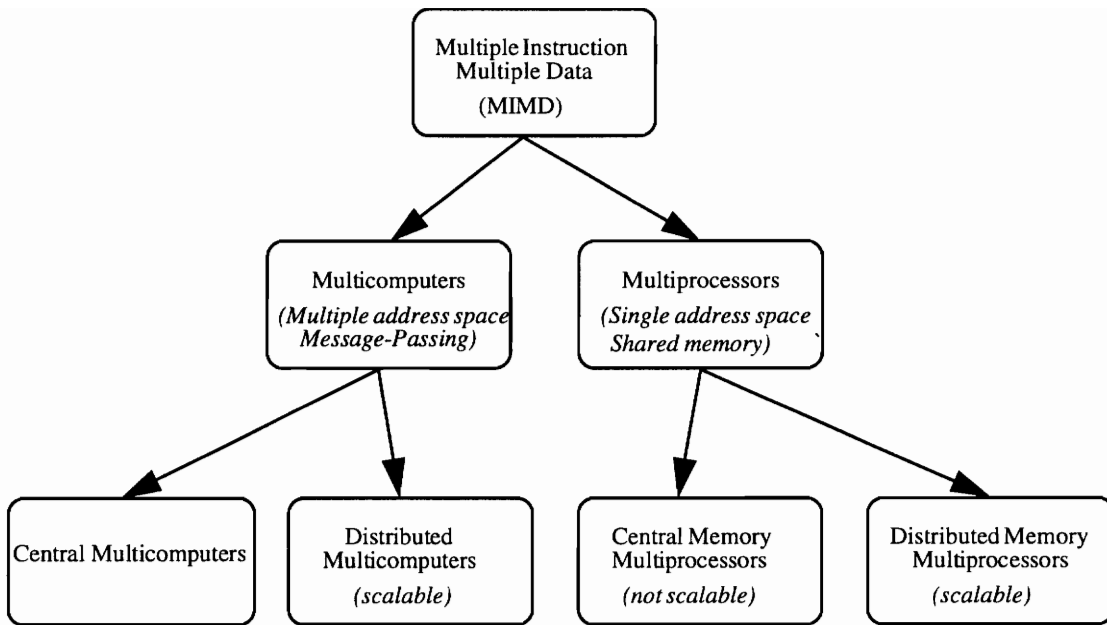


Each time a packet is forwarded through the network, additional packets of the message are transmitted. Wormhole routing, thus attempts to pipeline a message through the network at a grain size determined by the time required for routing at each intermediate node. If the packet encounters busy channels at the various intermediate nodes, it behaves like a packet switched network. On the other hand, if all the intermediate nodes are free, it behaves like a circuit switched network.

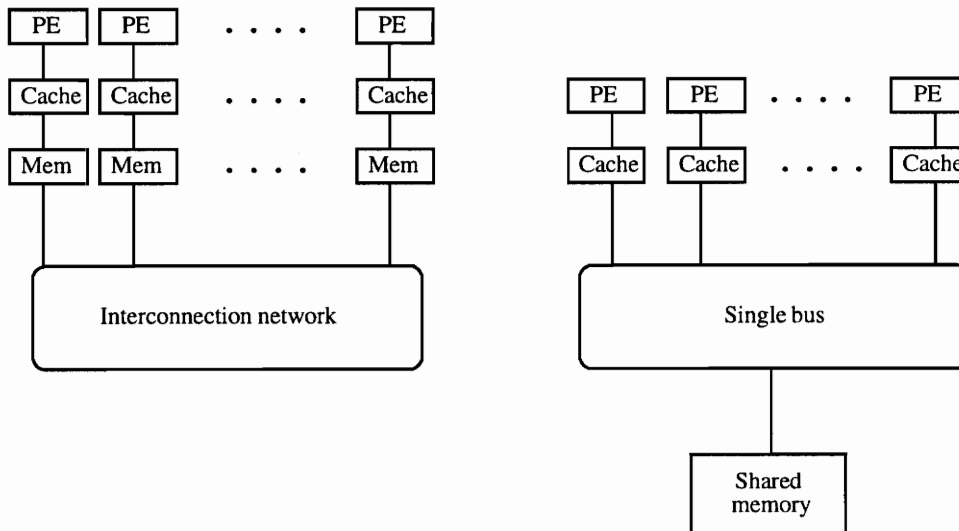
## **2.4 Communication and Synchronization Issues**

The issue of how processors should communicate data is very important since the method employed affects the programming model as well as the communication cost [9]. There is a class of machines which use a single address space, commonly known as multiprocessors. These machines use the shared-memory model which allows the processors to directly reference data from the memory which could be either centralized or distributed, based on the physical location of the memory with respect to the processors. Centralized shared memory models are typically used by small-scale bus based multiprocessing systems. To enable the shared memory model to be used with a scalable architecture, the memory has to be distributed independently of its logical sharing. This approach is known as the distributed shared memory model and is used by the Cray T3D machine which uses the 3-dimensional torus topology to connect the processors together [10].

Most of the large scale multicomputers built to-date have used the message-passing model for communication due to the fact that they are simpler to build than the shared memory model. However, with the message-passing model, the compiler (or programmer) has to partition a program into separate processes that communicate explicitly, by sending messages rather than implicitly through memory. Figure 2.3 and 2.4 show a break-up of the MIMD classification into multicomputers and multiprocessors [2].



**Figure 2.3** Taxonomy of MIMD Computers



**Fig 2.4** (a) MIMD system with distributed memory

(b) MIMD system with shared memory

### 2.4.1 Synchronization Issues in Shared Memory Computers

When a number of processors are acting concurrently, problems can arise when they try to read or modify the same data. A shared data object may be manipulated by operations that are implemented as sequences of events. If two processors concurrently perform such operations on the same data object, then one of them might be exposed to intermediate values produced by the other, and unpredictable results may occur. A concrete example would be in a situation where there is a stack or circular buffer that could be accessed simultaneously by two processors. One of the processors might modify the top of stack index and, before it has a chance to use this altered index to access data, the other processor could further modify the top of stack index resulting in the first processor producing incorrect results since it had unintentionally used the wrong data.

To prevent different processors from trying to access the same data simultaneously, a type of synchronization scheme known as “mutual exclusion” can be used. Mutual exclusion should be implemented in the “critical section” of the code so that access to the portions of code dealing with shared variables is given to only one competing process at a time. The problem with this method is that it can cause extensive delays when there are a large number of competing processes. The state when only one of the processors is in its critical section at any time is called “safe.” However, there is always the possibility of creating a circular wait state called “deadlock,” in which processes are waiting for events that cannot occur. The solution to the problem of mutual exclusion and deadlock is tricky. A state can occur where a process might wait forever for permission to enter its critical section and thereby “starve.” To prevent this situation from occurring, a method for mutual exclusion can be adopted in which prioritization of the processes takes place such that a process that has been waiting for a long time will get precedence over a process that has recently tried to access the shared variable.

The classic mutual exclusion algorithms generally use “busy waiting,” a method in which the competing processes for a critical section constantly seek to gain access to the section. Lamport’s Bakery algorithm solves the problem of starvation by prioritizing the processes in a first-come-first-served basis

when they try to access their critical section [11]. A problem with this method is that the prioritized queue could become arbitrarily large and cause an overflow.

The synchronization of parallel processes could be carried out by low-level primitives at the machine instruction level. Implementing atomic (indivisible) instructions in hardware significantly eases the software designers task. The exchange instruction (EXG) found in the Motorola 68000 family and the SWAP instruction found in the SPARC architecture use a set of three actions which are indivisible. Test and Set instructions are also indivisible. Hence, the synchronization is blocking in nature since it allows only one process to be active at one time, blocking the actions of all the others. This is therefore also the type of "busy waiting" mutual exclusion method discussed above.

Semaphores, a mutual exclusion scheme invented by Dijkstra [12], uses a pair of higher level primitives that operate on non-negative integer variables called semaphores. The entry and exit into the critical section are encapsulated by these high-level primitives, thereby disallowing the critical section to be interrupted by other processes through their operation.

#### **2.4.2 Synchronization Issues in Distributed Memory Multicomputers**

Distributed memory multicomputers consist of processor-memory pairs connected together by an interconnection network. Machines of this form typically have multiple address spaces and are commonly known as multicomputers. They use the message-passing model to communicate data between the processors. Machines like the mesh-connected Intel Paragon, TMCs fat-tree connected CM-5 and the hypercube connected network by NCUBE are examples of systems which use this form of communication [13]. In the case of message-passing computers, there is more self-synchronization than seen in communication through shared variables. If a message is not received for some reason after it has been sent or if it is overwritten by another message, then this creates a problem and the message has to be resent. In a message-passing parallel program, the constituent parallel parts or processes communicate via communication channels rather than through shared variables as is the case with shared-memory

systems. A channel provides a communication path between two processes and can be considered to be an abstraction of a physical communication link. In synchronous message-passing, each channel provides a direct link between two processes, and the sender blocks (or gets delayed) until the other process has received the message. The exchange of a message thus represents a synchronization point between the two processes.

## 2.5 Scalability of Parallel Processing Systems

*Scalability* is a common objective in the design of parallel processing systems. The term scalability has been interpreted in different ways by different people. People involved in algorithm design tend to view scalability in the context of the problem size, while people who work at the system level (hardware designers) tend to make the implicit assumption that the problem is scalable and then try to find an architecture on which to implement it. Thus the computer architect would define a scalable parallel computer to be one in which the interconnection scheme (and thus the number of processors) can be expanded without hurting the performance characteristic.

In 1967, Amdahl [14] put forth an argument which stated that even when the fraction of serial work in a given problem is small, say  $s$ , the maximum speedup obtainable from even an infinite number of parallel processors is only  $1/s$ . It can be clearly seen that in this argument the notion of scalability is closely tied to notion of speedup. With this point in mind, a scalable architecture would have to exhibit speedup linearly proportional to  $p$ , the number of processors used. However, this point of view does not cover the whole picture. The problem here is that the speedup could be of two types, simple speedup which keeps the problem size fixed and scaled speedup which allows the problem size to grow with the machine size.

In the first case, Amdahl's law is seen to hold true. It has been shown that even a small number of sequential operations can significantly limit the speedup achievable by a parallel computer. For example, if 10 percent of the operations have to be performed sequentially, then even with an infinite number of

processors, the time required for the overall operation of the problem cannot be reduced by more than a factor of 10. The validity of this argument led to research in the area of massively parallel processing being in a dormant state for a very long time.

It was only after the problem was looked at from a different perspective, that there was renewed interest in this area. It was seen that when a faster computer was provided to solve a problem, it was usually not used just to solve the problem faster, but instead to solve a larger problem in roughly the same amount of time. Thus, scaled speedup allows the problem size (as measured by its serial complexity) to increase linearly with the number of processors. Because the opportunity for parallelism tends to grow with the size of the problem, the percentage of time required for the operations to be done in parallel tends to grow more rapidly than the serial time. A problem may grow either because a more complex system is being analyzed or because the same system is being analyzed to a higher degree of accuracy.

### **2.5.1 Network scaling factors**

There are several factors which are used to determine how performance characteristics scale with the number of processors being interconnected using a specific network topology. A network can be represented by the graph of a finite number of nodes linked by directed or undirected edges [2]. The number of nodes in a graph is called the network size. Some of the factors are as follows:

**2.5.1.1 Node Degree** -- The number of edges (links or channels) incident on a node is called the node degree. It reflects the number of I/O ports required per node and helps in determining the cost of the node. Thus, the node degree should be kept as small as possible. A constant node degree is highly desirable in order to achieve modularity in building blocks for scalable systems.

**2.5.1.2 Network Diameter** -- This is the maximum number of communication links that must be traversed to transmit a message between any source/destination pair along a shortest path. It places an upper bound

on the delay required to propagate information throughout the network. It should be as small as possible to enable efficient communication through the network.

**2.5.1.3 Bisection Width** -- When a network is cut into two halves, the minimum number of edges (channels) along the cut is called the channel bisection width. It is used to help provide an indication about the maximum communication bandwidth along the bisection of a network.

Table 2.1 shows how the performance characteristics of interconnection networks scale with the number of processors being connected. The hypercube topology has high connectivity and a low network latency through the connection of each of the  $N$  processing elements to  $\log N$  neighbors. It can be seen that the wire cost increases by  $\log N$  while the latency is reduced by  $\log N / \sqrt{N}$ . The mesh topology is relatively inexpensive, but the network diameter increases by a factor of  $\sqrt{N}$ , a high rate of increase, which is not very suitable since the maximum delay of a message from one of  $N$  processors to another should be as small as possible. The tree topology appears to be a good compromise, as its latency is the same as that of a cube, while its wire cost is the same as that of a mesh since its node degree is independent of  $N$ . In fact, fat tree networks are considered “universal” in the sense that they can efficiently simulate any other network of the same volume [15]. In the table shown, networks with a higher performance are shown towards the top while networks which cost less are shown towards the bottom [7].

## 2.6 Summary

In this chapter, an overview of multicomputer networks was presented. A classification of parallel processing architectures was provided along with performance criteria and design issues related to the various interconnection network topologies. Communication and synchronization issues of MIMD systems were analyzed. Scalability problems were discussed with respect to Amdahls Law and the factors affecting the scalability of parallel processing systems were enumerated. In the next chapter, various design issues related to the development of a single processing node, used in a message-passing multicomputer network, are addressed in detail.

**Table 2.1 Network Scaling Factors** (Interconnection network tradeoffs showing how performance characteristics scale with the number of processors (N) being connected).

Network type	Node Degree	Network Diameter	Number of links	Bisection Width	Connectivity	Wire Cost
Linear Array	2	$N - 1$	$N - 1$	1	2 nearest neighbors	$N$
Star	$N - 1$	2	$N - 1$	$\lfloor N/2 \rfloor$	1	$N$
Ring	2	$\lfloor N/2 \rfloor$	$N$	2	2 nearest neighbors	$N$
2-D Mesh	4	$\sqrt{N}$	$2N - 2\sqrt{N}$	$\sqrt{N}$	4 nearest neighbors	$N$
Binary Tree	3	$\log N$	$N - 1$	1	3 nearest neighbors	$N$
Completely connected	$N - 1$	1	$N(N - 1)/2$	$(N/2)^2$	any to any	$N^2$
Hypercube	$\log N$	$\log N$	$N \log N / 2$	$N/2$	$\log N$ nearest neighbors	$N \log N$
2-D Torus	4	$2\lfloor \sqrt{N/2} \rfloor$	$2N$	$2\sqrt{N}$	4 neighbors	$N$



## **Chapter 3**

### **Design of A Single Processing Node**

#### **3.1 Introduction**

In the early 1980's, microprocessor performance became a major research area with focus on quantitative metrics such as instruction execution rates and instruction counts. Sophisticated architectural techniques, which had been in widespread use in large mainframes and supercomputers, were incorporated into the microprocessors. For example, pipelining and high-performance memories had been used by supercomputers for many years. The introduction of RISC processors with a simple instruction set made it easier to migrate these techniques to the microprocessor.

This chapter describes the VHDL model of the processing nodes which will be used in the fat-tree multicomputer network. The structural and behavioral level VHDL models of the processing node are based on the DLX processor, a 32-bit reduced instruction set processor detailed in [1]. The reasons for choosing a RISC-based processor are pointed out. The instruction set of the DLX processor is described, along with the instruction format used by the processor. The efficient use of the memory hierarchy by parallel algorithms is considered in the design, with the result that each processing node has an individual cache memory included in it. In Section 3.9, the use of instruction level parallelism to decrease the cycles per instruction (CPI) of the processor and thereby improve its performance is explored with a suitable test-bench program.

## 3.2 Evolution of RISC Microprocessor Architectures

In the mid-seventies, studies were carried out by computer architects to study the frequency of the various instructions performed by complex computers of the time. It was observed that even the most sophisticated computers executed mostly simple instructions. The RISC-II microprocessor, designed at U.C. Berkeley in 1984, was a 32-bit microprocessor with 138 registers and a 330 ns cycle time. The MIPS design, initiated by Hennessey, was a 32-bit microprocessor with 16 registers and a 500 ns cycle time. The Swift (Simulation of Widely Interconnected Fat-Trees) processor is an alias for the DLX processor [1], which is similar to the commercially available MIPS R2000 processor. Table 3.1 shows a comparison of the major features of some of the older complex instruction set computer (CISC) architectures. There were wide differences between the microprocessor architectures in their instruction size, data alignment, data addressing modes, page size and integer registers. Table 3.2 shows a comparison of the same set of features in the case of the more recent load/store architectures which are also known as reduced instruction set computer (RISC) architectures. In this table, there is a significant difference noticed in comparison to Table 3.1, with many of the features being identical in the various processors examined. Furthermore, RISC processors have fewer and simpler types of instructions than CISC processors and, consequently, a RISC instruction requires less processing logic to interpret than a CISC instruction. The higher instruction execution rate of RISC processors with their simple instruction sets and optimized software primitives are important reasons for the present trend towards RISC processors in parallel processing systems. In this thesis, the DLX processor has been chosen as the processing element to be included in the fat-tree multicomputer due to its features being very similar to those found in some of the commercially available reduced instruction set processors, as seen in Table 3.2. Some of the features seen most often in RISC-based processors are the following:

- Single cycle execution -- Most instructions are executed in a single machine cycle.
- Hardwired control with little or no microcode -- Microcode adds a level of complexity and raises the number of the cycles per instruction (CPI).

**Table 3.1** Comparison of the major features of some of the older CISC architectures

	<b>IBM 360/370</b>	<b>Intel 8086</b>	<b>M68000</b>	<b>DEC VAX</b>
Date announced	1964/1970	1978	1980	1977
Instruction size (bits)	16,32,48	8,16,24,...,48	16,32,48,64,80	8,16,24,...,432
Data alignment	Yes/No	No	Yes (16-bit aligned)	No
Data addressing modes	4	5	9	$\geq 14$
Page size	4 KB	--	.25 to 32 KB	0.5 KB
Integer registers (size, model, number)	16 GPR $\times$ 32 bits	8 dedicated data $\times$ 16 bits	8 data and 8 address $\times$ 32 bits	15 GPR $\times$ 32 bits

**Table 3.2** Comparison of the major features of some of the recent RISC architectures (from [1])

	<b>Swift (DLX)</b>	<b>Intel i860</b>	<b>MIPS R2000</b>	<b>SPARC</b>
Date announced	1990	1989	1986	1987
Instruction size (bits)	32	32	32	32
Data alignment	Aligned	Aligned	Aligned	Aligned
Data addressing modes	1 (16-bit displacement)	2 (16-bit displacement, indexed mode)	1 (16-bit displacement)	2 (13-bit displacement, indexed mode)
Page size	4 KB	4 KB	4 KB	4 KB
Integer registers (size, model, number)	31 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits	31 GPR $\times$ 32 bits

- Load/Store, register to register design -- All computational instructions involve registers. Memory accesses are made with only load and store instructions.
- Simple fixed-format instructions with few addressing modes -- All instructions are the same length (typically 32-bits) and have just a few ways to address memory.
- Pipelining -- The instruction set allows for the processing of several instructions at the same time.
- High performance memory -- RISC machines usually have at least 32 general-purpose registers and large cache memories.

### 3.3 Instruction Set Architecture of the 32-bit DLX Processor

The DLX processor has thirty-two 32-bit general-purpose registers (GPRs). By convention, the value of register R0 is zero and it is a read-only register. All memory references are through loads or stores between memory and the general purpose registers. Accesses involving the GPRs can be to a halfword, or a word, and any of the GPRs may be loaded or stored, except that loading R0 has no effect. All ALU instructions are register-register instructions. The operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR, and shifts.

There are also compare instructions, which compare the contents of two registers ( $=$ ,  $\neq$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ). If the condition is true, these instructions place a 1 in the destination register; otherwise they place the value 0. Since a register is 'set' by these operations, they are called set-equal, set-not-equal, set-less-than and so on.

Control is handled through a set of jumps and a set of branches. The jump instructions are differentiated by the two ways to specify the destination address and by whether or not a link is made. All branches are conditional. The branch condition is specified by the instruction, which may test the register source for zero or non-zero. The value found in the register could be a data value or the result of a compare. The branch target address is specified with a 16-bit signed offset that is added to the program counter. Table 3.3 shows a list of the instructions used by the processor. The instruction type is followed

**Table 3.3** A list of the instructions used by the Swift (DLX) processor model - Load and store instructions, arithmetic/logical instructions, and control-flow instructions.

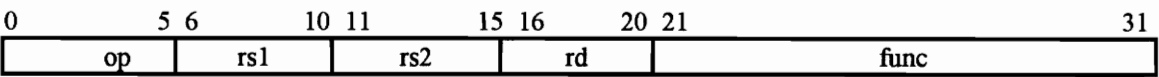
Instruction type/opcode	Instruction name	Example Instruction	Instruction meaning
LH, SH	Load halfword, store halfword	LH R1,40(R3) SH 2(R2),R3	$R1 \leftarrow \text{32}(M[40+R3])_{16}$ $\text{32}(M[41+R3]) \leftarrow \text{16} R3_{16,31}$
LW, SW	Load word, store word	LW R1,10(R0) SW 5(R4),R3	$R1 \leftarrow \text{32} M[10+0]$ $M[5+R4] \leftarrow \text{32} R3$
ADD, ADDI	Add, add immediate	ADD R1,R2,R3 ADDI R1,R2,#3	$R1 \leftarrow R2 + R3$ $R1 \leftarrow R2 + 3$
SUB, SUBI	Subtract, subtract immediate	SUB R1,R2,R3 SUBI R1,R2,#3	$R1 \leftarrow R2 - R3$ $R1 \leftarrow R2 - 3$
AND, ANDI	And, and immediate	AND R1,R2,R3	$R1 \leftarrow R2 \& R3$
OR, XOR	Or, exclusive or,	OR R1,R2,R3	$R1 \leftarrow R2   R3$
LHI	Load high immediate	LHI R1, #42	$R1 \leftarrow 42\#\#0^{16}$
SLL, SRL, SRA	Shift left logical, shift right logical, shift right arithmetic	SLL R1,R2,#5	$R1 \leftarrow R2 \ll 5$
SLT, SGT, SEQ,SNE	Set less than, set greater than, set equal to, set not equal to	SLT R1,R2,R3	if ( $R2 < R3$ ) $R1 \leftarrow 1$ else $R1 \leftarrow 0$
BEQZ, BNEZ	Branch GPR equal to zero, branch GPR not equal to zero	BEQZ R4, name	if ( $R4 == 0$ ) $PC \leftarrow \text{name};$ $((PC+4) - 2^{15}) \leq \text{name} < ((PC+4) + 2^{15})$
J, JR	Jump, jump register	JR R3	$PC \leftarrow R3$
JAL, JALR	Jump and link, Jump and link register	JALR R2	$R31 \leftarrow PC + 4;$ $PC \leftarrow R2$

by the instruction name and an example instruction has been provided along with its meaning to make it easy to understand.

### 3.4 Instruction Format

The layout of the instruction word is called the instruction format. The selection of an instruction set requires a delicate balance between the number of instructions needed to execute a program, the number of clock-cycles needed by an instruction, and the speed of the clock. The instruction set is divided into three types of instruction formats. They are as follows:

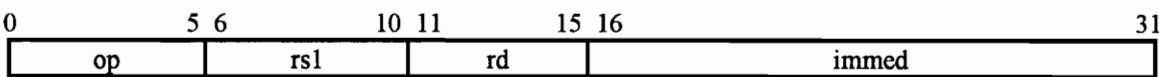
The R-type (register) instruction is exactly 32 bits in length (the same size as that of a data word) and is used for ALU operations. The fields of this format are:



The op field is the opcode, rs1 and rs2 are the source register addresses, rd is the destination register address and the func field selects the variation of the operation in the op field.

$$rd \leftarrow rs1 \text{ func } rs2$$

The I-type (immediate) instruction is used primarily for data transfer operations. The fields of this format are:



The op field is the opcode, rs1 is the source register address, rd is the destination register address and the immed field is an immediate displacement.

$$rd \leftarrow rs1 \text{ op immed}$$

It can be observed that the formats of the R-type and I-type instructions are similar. The first three fields of these two formats have the same length with the fourth field in the I-type instruction being equal in length to the last two fields of the R-type instruction. The first field (the opcode), is used to distinguish between these two formats. Each format is assigned a set of values in this field so that the hardware

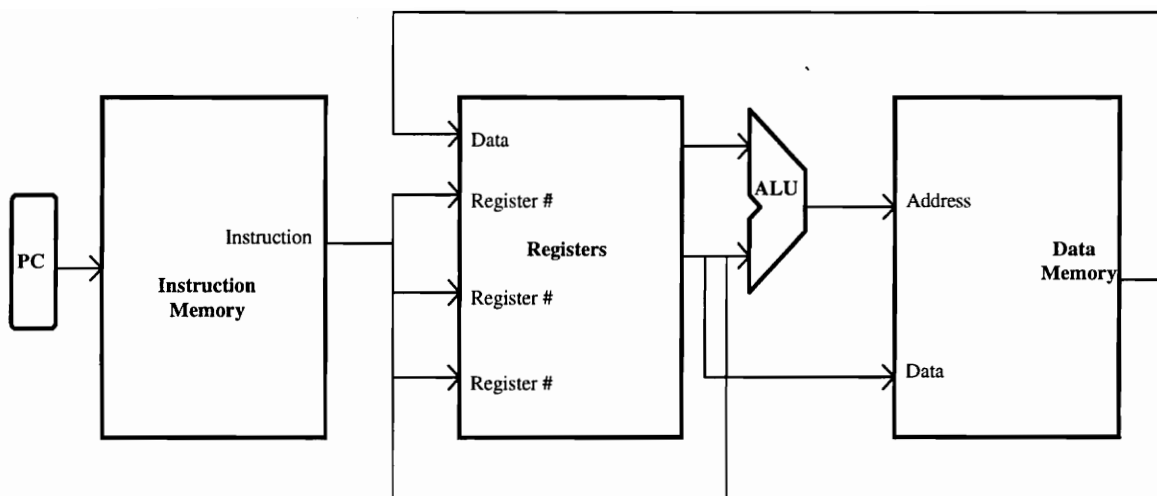
knows whether to treat the last half of the instructions as two fields (R-type) or as a single field (I-type). The conditional branch instructions use the I-type instruction format.

The J-type (jump) instruction format is the simplest of the three types. It consists of the first 6 bits used for the opcode and the rest of the bits are used for the address field and can be used for both jump and jump-and-link instructions.

### **3.5 Basic Steps of Execution**

The processor contains the hardware components for processing instructions and data. It consists of a control unit and a datapath which together supervise and implement the various data processing tasks. The control unit is the part that effects the retrieval of instructions in proper sequence, the interpretation of each instruction, and the application of the proper signals to the arithmetic units and the other parts.

All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by the instruction are determined by examining the appropriate fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (from an integer arithmetic-logic instruction), or a compare (branch). If the instruction is an arithmetic-logic instruction, the result from the ALU must be written to the result register. If the operation is a load or a store, the ALU result is used as an address to either store a value from the register or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which requires some control logic. Figure 3.1 displays an abstract view of the implementation of the processor subset. For the processor instruction set, all instructions can be broken down into five basic steps as follows:



**Figure 3.1** An abstract view of the implementation of the processor subset, showing the major functional units and the connections between them.



#### i) Instruction fetch step

$MAR \leftarrow PC$  -- Send the program counter (PC) to a memory that contains  
 $IR \leftarrow M[MAR]$  -- the code to fetch the instruction

#### ii) Instruction decode/register fetch step

$A \leftarrow Rs1$  -- Decode the instruction and access the register file to  
 $B \leftarrow Rs2$  -- read the instructions  
 $PC \leftarrow PC + 4$  -- Increment the PC to point to the next instruction

#### iii) Execution/effective address step

The ALU operates on the operands, performing one of three functions depending on the instruction type (as discussed earlier) :

- **Memory reference**

$MAR \leftarrow A + (IR_{16})^{16} \# IR_{16..31}$  -- The ALU adds the operands to form the effective address  
 $MDR \leftarrow Rd$  -- The memory data register (MDR) is loaded for a store

- **ALU Instruction**

$ALU \leftarrow A \text{ or } (B \text{ or } (IR_{16})^{16} \# IR_{16..31})$  --The ALU performs the operation specified by the opcode  
-- on the value in A and on the value in B or the sign-  
-- extended immediate

- **Branch/Jump**

$ALU \leftarrow A \text{ or } (B \text{ or } (IR_{16})^{16} \# IR_{16..31})$  -- The ALU adds the PC to the sign-extended immediate  
 $Cond \leftarrow (A \text{ op } 0)$  -- value to compute the address of the branch target

#### iv) Memory access/branch completion step

A memory reference instruction will need to access the memory containing the data to complete a store or get a word that is being loaded. An arithmetic logic instruction must write the data from the ALU back into a register.

$MDR \leftarrow M[MAR]$  -- If the instruction is a load, data returns from memory

or  $M[MAR] \leftarrow MDR$  -- If the instruction is a store, data is written to memory  
 If (cond)  $PC \leftarrow ALU_{output}$  (branch) -- If the instruction branches, the PC is replaced with the  
 -- branch destination address

#### v) Write-back step

$RD \leftarrow ALU_{output}$  or  $MDR$  -- Write the result into the register file, whether coming from  
 -- the memory or from the ALU

### 3.6 VHDL Model of the Datapath

The datapath within the DLX computer consists of the execution units such as the arithmetic logic units, the registers, and the communication paths between them. Figure 3.2 shows a block diagram of the major functional units and the major connections between them. The paths for control are seen from the control unit to the ALU and the various registers. The paths for data transfer are seen between the S1 bus, S2 bus, Dest bus and the various components. The processor uses three buses: S1, S2 and Dest. The only path from the S1 and S2 buses to the Dest bus is through the ALU. It has two operand input ports, a result output and condition code output ports.

The behavior of the ALU has been implemented using a VHDL process statement, which is sensitive to changes on the operand and command input ports [16]. There are three condition code (CC) register bits which are updated after each arithmetic or logical instruction. The zero bit is set if the result is zero. The negative bit is set if the result of an arithmetic instruction is negative, and is undefined after logical instructions. The overflow bit is set if the result of an arithmetic instruction exceeds the bounds of representable integers, and is also undefined after logical operations.

The register file consists of a set of registers that can be read and written by applying a register number to be accessed. From Figure 3.2 it can be seen that there are two read ports with two data bus outputs (q1 and q2), and the write port has a data input (d3). To write a register, three inputs need to be specified: a register number, the data to be written and a clock that controls the writing to the register.

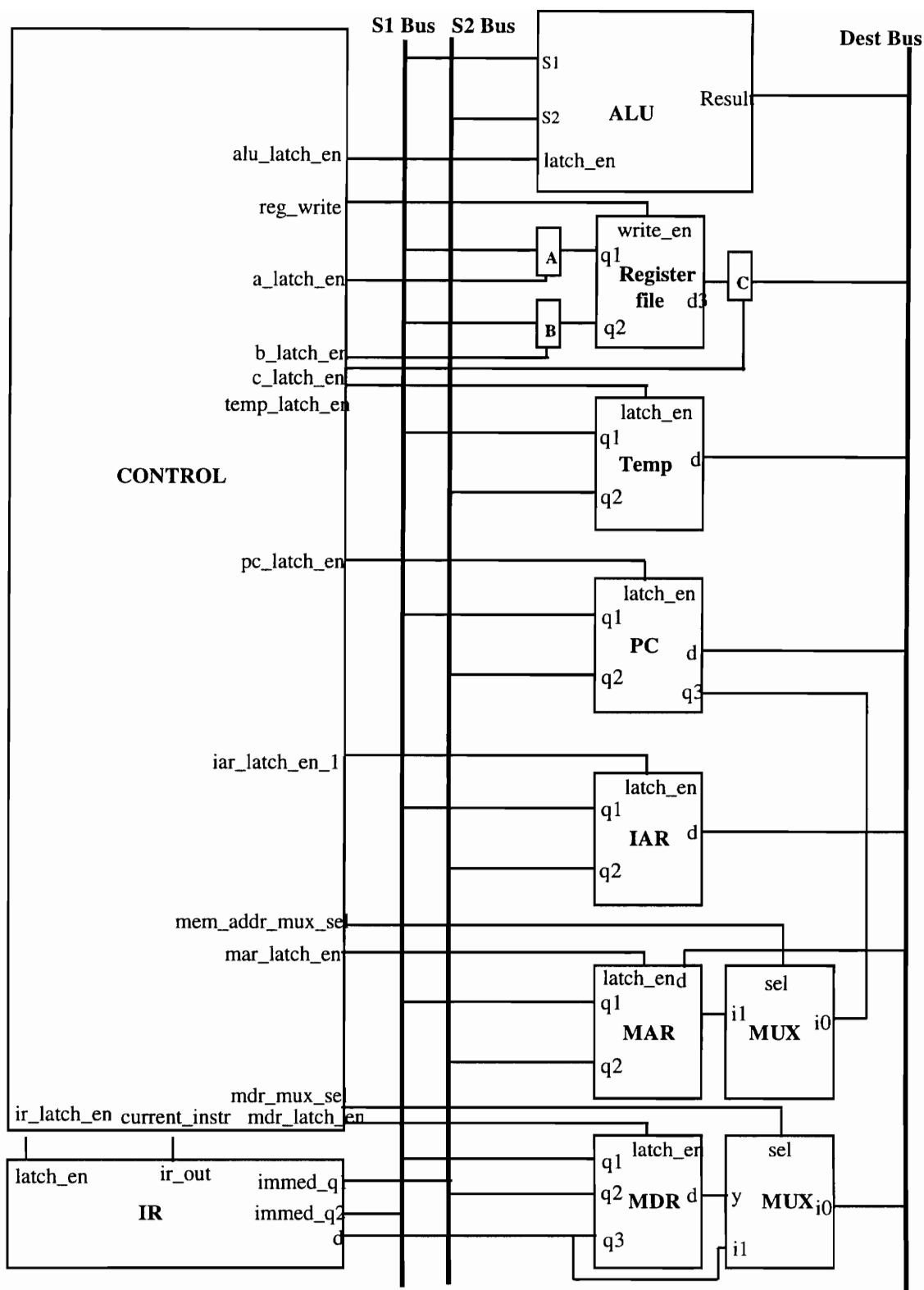


Figure 3.2 Block diagram of the 32-bit DLX processor

The behavior of a register is implemented with a VHDL process statement. When any of the inputs change, the write port enabled is checked first, and if asserted, the addressed data is fetched, and driven onto the corresponding data output bus. If the port is disabled, the data output bus driver is disconnected.

The function of a latch is to store a signal with the output being equal to the value of the stored state inside the element. It has an enable input bit, `latch_en`, a bit vector input `d`, and a bit vector output `q`. When `latch_en` is '1,' changes on `d` are transmitted through to `q`. When `latch_en` changes to '0,' any new value on `d` is ignored, and the current value on `q` is maintained. In the datapath shown in Figure 3.2, there are latches on the two outputs of the register file (A and B) and a latch on the input (C).

A multiplexer performs the function of selecting as its output, one of its inputs, that is specified by a control. A two-input multiplexer could be specified with a select input bit, two bit vector inputs `i0` and `i1`, and a bit-vector output `y`. A VHDL concurrent signal assignment statement, which uses the value of the select input to determine which of the two bit-vector inputs is passed through to the output.

The other registers which are part of the state of the machine are the program counter (PC) and the interrupt address register (IAR). The PC is a register used to hold the address of the current instruction being executed. The memory address register (MAR), memory data register (MDR) and temporary registers are used in the execution of instructions.

### 3.7 Processor Bus Architecture

The processor communicates with its memory over synchronous 32-bit address and data buses. The two clock inputs, `phi1` and `phi2`, provide a two-phase non-overlapping clock for the processor. Each cycle of the `phi1` clock defines one of three bus states: `Ti` (idle), `T1` or `T2`. Bus transactions consist of a `T1` state followed by one or more `T2` states, with `Ti` states between transactions.

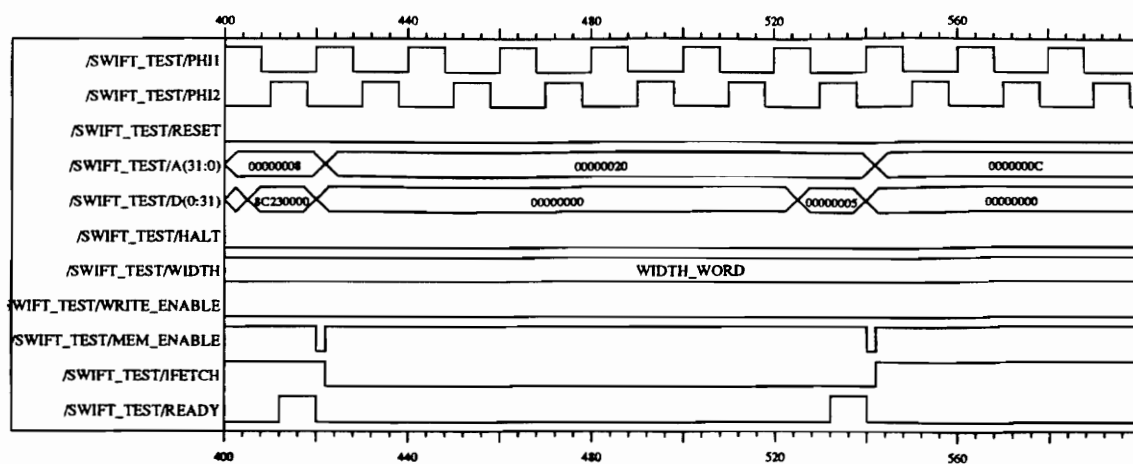
During an idle state `Ti`, the processor places the memory address on the address bus to begin the transaction. The next state is a `T1` state. After the leading edge of the `phi1` clock, the processor asserts

the `mem_enable` control signal, indicating that the address is valid and the memory should start the read transaction. The processor also asserts the `ifetch` signal if it is reading instructions. It always leaves the `write_enable` signal negated during read transactions. Figure 3.3 shows a memory read operation. The value to be read by the processor, (5), is put on the data bus at address 20. It can be seen that the `ifetch` signal is negated after the data has been read by the processor. Figure 3.4 shows a memory write operation. `Write_enable` is seen to be asserted at the time that the value to be written, (C), is put on the data bus at address 24. This value has been obtained by adding the number 5 and 7 to obtain C (Hex). After the data has been written by the processor, the `write_enable` signal is negated to indicate that the transaction is complete. At the end of the transaction there is a null signal assignment to the data bus port, indicating that the processor is disconnecting itself from the data bus. In the next section, the reasons for including a local cache memory in each processing node of a multicomputer are discussed.

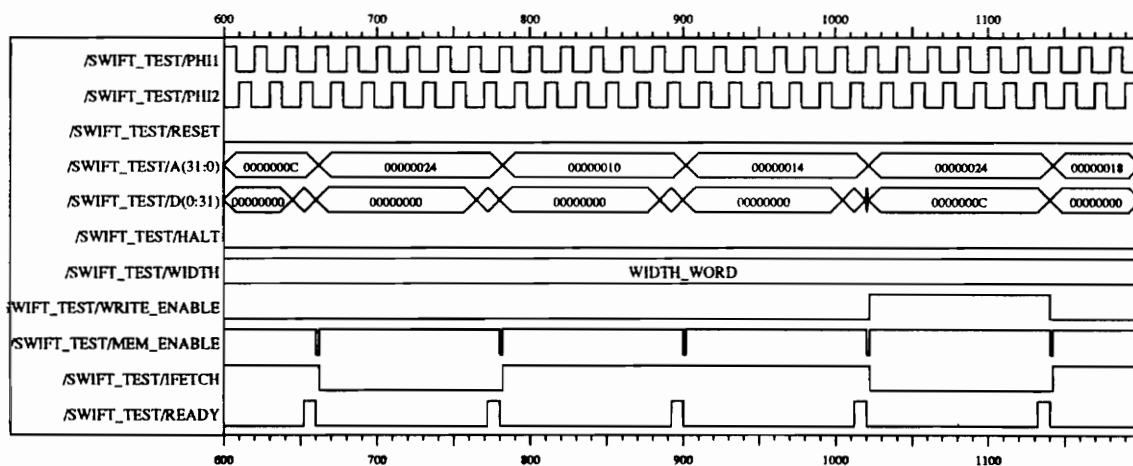
### 3.8 Cache Design Issues

Traditional memory systems have been implemented with dynamic RAMs (DRAMs). While memories have gotten faster in the last few years, the increase in speed of DRAM memories has not kept pace with the increase in speed of microprocessors. The result is that data and code requests from the processor to DRAM main memory cause the processor to incur wait states, thereby lowering performance. Small cache memories using static RAM (SRAM) have provided the best way for system designers to maximize performance while maintaining reasonable system cost. The system is configured so as to enable the processor to spend the majority of time operating out of the small cache memory.

In scientific and engineering applications, the access patterns for instructions and data are fairly specialized, which allows optimization of the memory system accordingly. For example, for many scientific applications, the program size is much smaller than the data size [17]. Furthermore, the program exhibits high locality of reference which means that only a small portion of the program is heavily used during any given interval. This locality, called *temporal locality*, arises because the program



**Figure 3.3** Processor bus read transaction



**Figure 3.4** Processor bus write transaction

often consists of several nested loops that execute the same instructions many times. Temporal locality can be exploited by keeping recently accessed instructions in a cache, thereby making efficient use of the memory hierarchy. There is often temporal locality found in data accesses, even with very large data sets.

Another form of access observed in instruction and data accesses is *spatial locality*. Spatial locality refers to the tendency to make use of instructions or data elements that are close together in memory at the same time. Spatial locality can be exploited by retrieving memory words that are close to a word that is requested in parallel with the requested word, with the hope that the processor will need the nearby words soon.

These locality properties do not come from any inherent property of computation but are the result of extensive observations on how programs behave [3]. The use of cache memories in the individual processing nodes of multicomputer networks can lead to low latency of access. Vector supercomputers on the other hand do not usually contain caches. Instead, most vector supercomputers contain a small set of low-latency vector registers. These machines are designed to move an arbitrary vector from memory to the CPU as fast as possible. Moving an entire vector makes use of spatial locality. However, since there are not many vector registers (8 to 64 typically), only a small number of vectors can be kept close to the CPU at any time.

Thus, there is an important difference observed between the operation of multicomputers and the operation of vector supercomputers. The multicomputer processors efficiently use the cache memory to provide lower average access latency. Because the vector registers are relatively small, many data accesses in a vector machine need to go to the main memory. To minimize the long latency of memory access in a vector supercomputer, many such machines construct their main memory from SRAM. This reduces the access latency, but at a very high cost. Thus, memory systems in vector supercomputers are normally an order of magnitude more expensive per bit than the memory systems of microprocessor-based machines with cache-based memory hierarchies. Figure 3.5 shows a block diagram of the processor with cache and memory modules and the interconnections between them. A `cpu_cache_monitor` unit and a

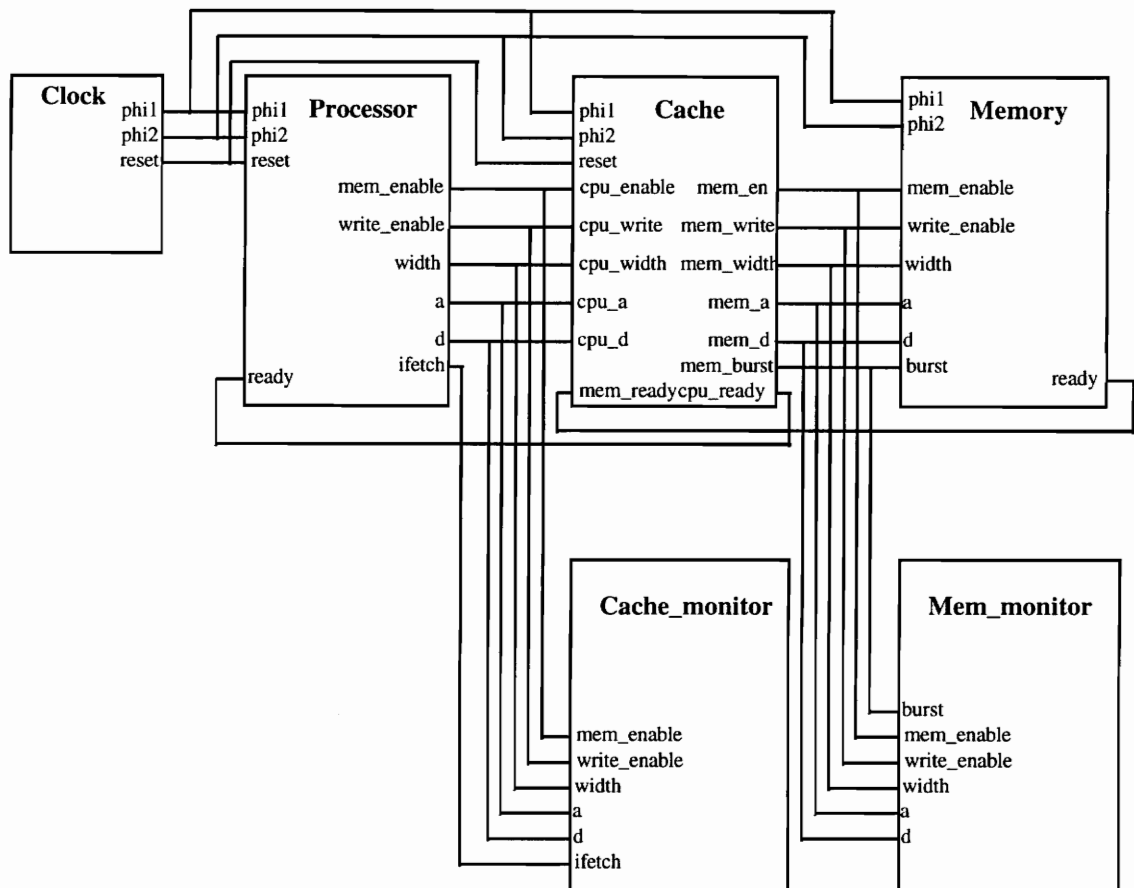
cpu\_mem\_monitor unit have been added to help monitor the state of the buses at any given instant of time.

An important issue is the cache *write strategy*. There are two basic techniques used to write to a cache. The information can be written to the block in the cache as well as to the block in the lower level memory. This can be seen in Figure 3.6 where the mem\_write signal is also enabled when the cpu\_write signal is enabled. This is known as the write-through strategy. The other write policy is known as write-back because the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced. Figure 3.7 shows the output obtained when using such a cache strategy. The mem\_write signal is not enabled, even though the cpu\_write signal has been enabled, indicating that the cache is being written to but the memory is not.

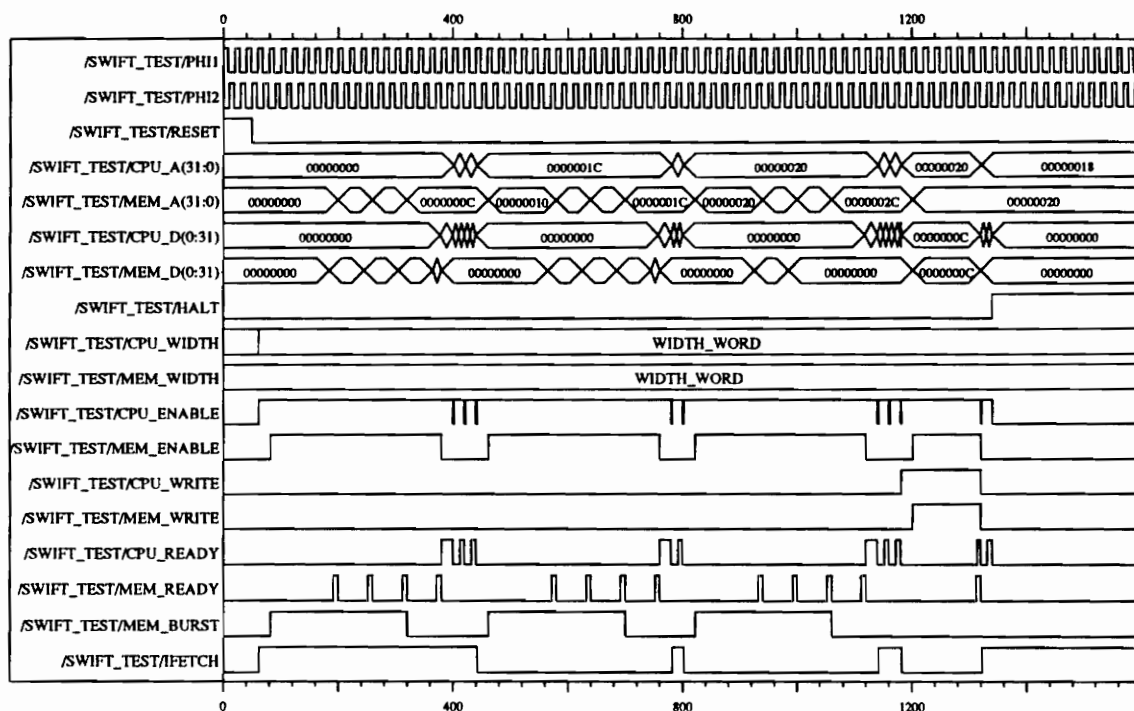
Chip designers have to make decisions on chip design that could have important consequences in the future which were unanticipated. It is important to decide what kind of applications a chip is being designed for. For example, the Intel family of CISC microprocessors ranging from the 80286 to the 80486 were designed to be general purpose processors for desktop computing. The Intel i860 has been designed with a two-way set associative cache (which views main memory as being logically broken up into two pages). This decision was based on the assumption that the i860 was going to be used as a graphics coprocessor and not as a general purpose CPU. The 80486 was designed with a 4-way set associative cache as it was found to reduce the miss rate compared to that of a 2-way cache by 10% [3]. Based on the fact that the current generation of multicomputers are being designed to support a wide variety of general purpose applications, the inclusion of a 4-way cache in each processing node is a more suitable design choice.

In this section, we have examined ways to improve the average memory access time of a processor with the use of a cache memory. Cache design issues were discussed with the help of sample test programs simulated on the DLX processor.





**Figure 3.5** Block diagram of the SWIFT processor with cache and memory modules



**Figure 3.7** Processor write to cache (write-through)

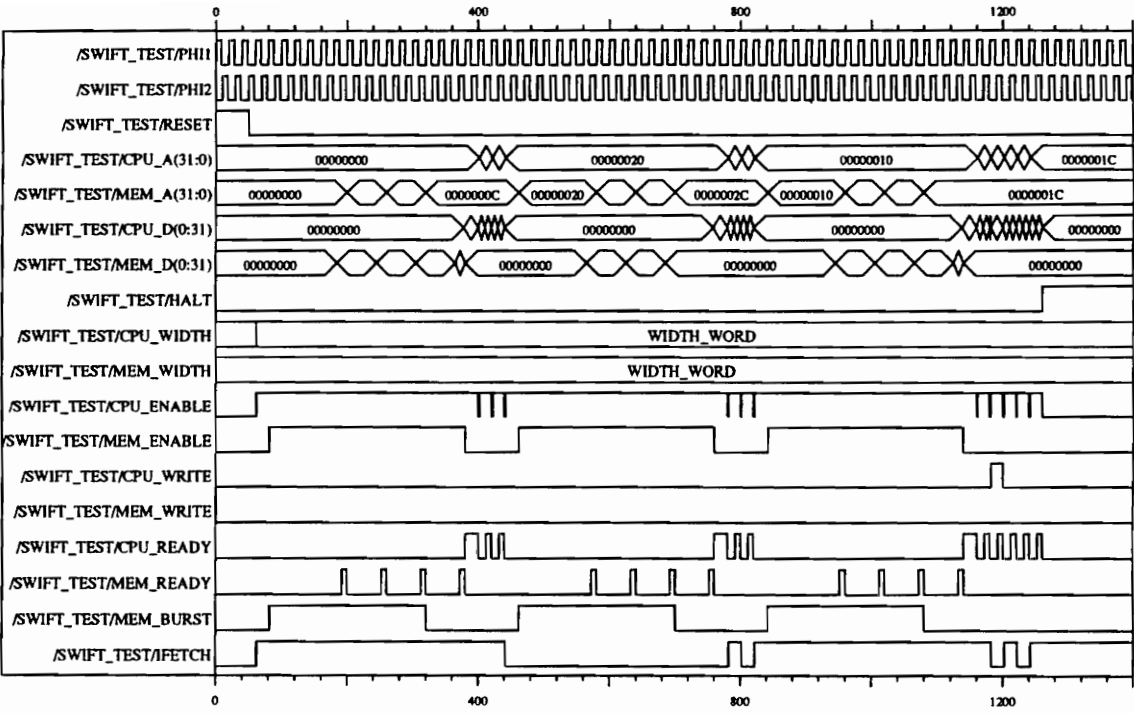


Figure 3.7 Processor write to cache (write-back)

### 3.9 Instruction-Level Parallelism

In this section, a technique used to improve the performance of the processor by overlapping multiple instructions in a "pipeline" is explored. There are many techniques which can be used to execute more instructions per clock cycle and thereby improve the performance of a single processor. Instruction-level parallelism can be exploited by finding sequences of unrelated instructions that can be overlapped in a pipelined processor. The example shown below is used to evaluate the approach of improving instruction level parallelism through a technique known as *loop unrolling*. Loop unrolling is performed by initially replicating the loop body a multiple number of times. After this step, the loop termination code is adjusted and then the unrolled loop is *scheduled* by interchanging the loads and stores which are found to be independent of one another. The program given below, is written in assembly language, using the instruction format for the DLX processor seen in Table 3.3. The assembly language files are assembled into a memory image by an instruction set assembler. In the program, a simple loop is used, which adds a scalar value to a vector in memory.

```
; This program calculates x[i] = x[i] + a
.data 0
    ADD    R3, R0, a
Start: LW    R2, 0(R3)
    ADD    R1, R0, xtop
Loop: LW    R5, 0(R1)
    ADD    R4, R5, R2
    SW     0(R1), R4
    SUB    R1, R1, #4
    SLT    R6, R1, #40
    BEQZ   R6, Loop
    TRAP   #0

x:    .word 0,1,2
xtop: .word 3
a:    .word 3
```

Figure 3.8 shows the output of the test bench. The total time to add the scalar value to the vector is approximately 4000 ns of simulation time with each loop iteration taking about 1000 ns. A simple way to increase the number of instructions between executions of the loop branch is through the use of loop

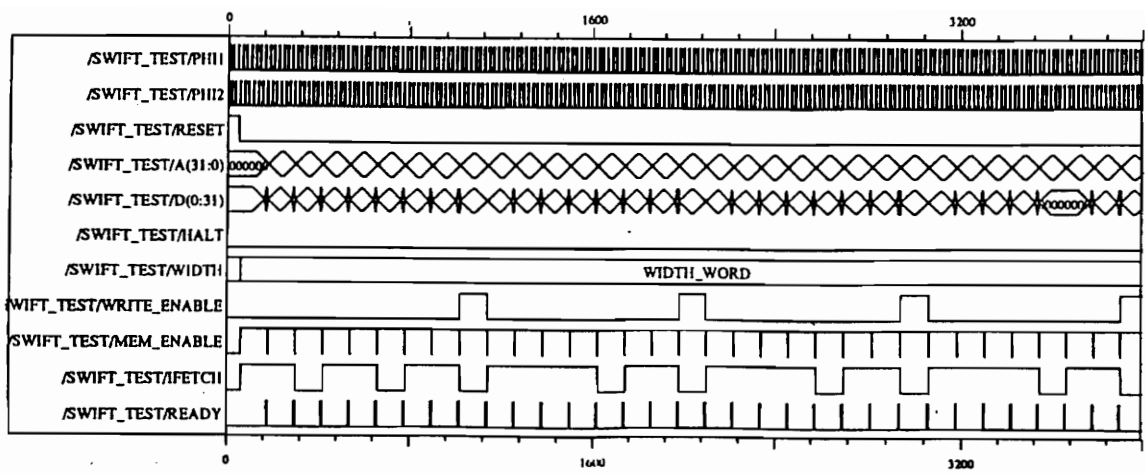
unrolling. The loop body is replicated a number of times based on the vector length, and then it is scheduled. In order to schedule the unrolled loop, it is noticed that two instructions, SUB and SW, can be interchanged, by changing the address that SW stores to, from 0(R1) to 12(R1). This change allows the elimination of three branch instructions and three decrements of R1. If the loop from the above example is unrolled 3 times and then scheduled after checking for dependencies, the code would be the following:

```
; This program uses a scheduled, unrolled loop to calculate  $x[i] = x[i] + a$ 
.data 0
    ADD    R3, R0, a
Start: LW   R2, 0(R3)
    ADD    R1, R0, xtop
Loop: LW   R5, 0(R1)
    LW     R6, -4(R1)
    LW     R10, -8(R1)
    LW     R14, -12(R1)
    ADD    R4, R5, R2
    ADD    R8, R6, R2
    ADD    R12, R10, R2
    ADD    R16, R14, R2
    SW     0(R1), R4
    SW     -4(R1), R8
    SW     -8(R1), R12
    SUB    R1, R1, #22
    SLT    R7, R1, #72
    BEQZ   R7, Loop
    SW     12(R1), R16
    TRAP   #0

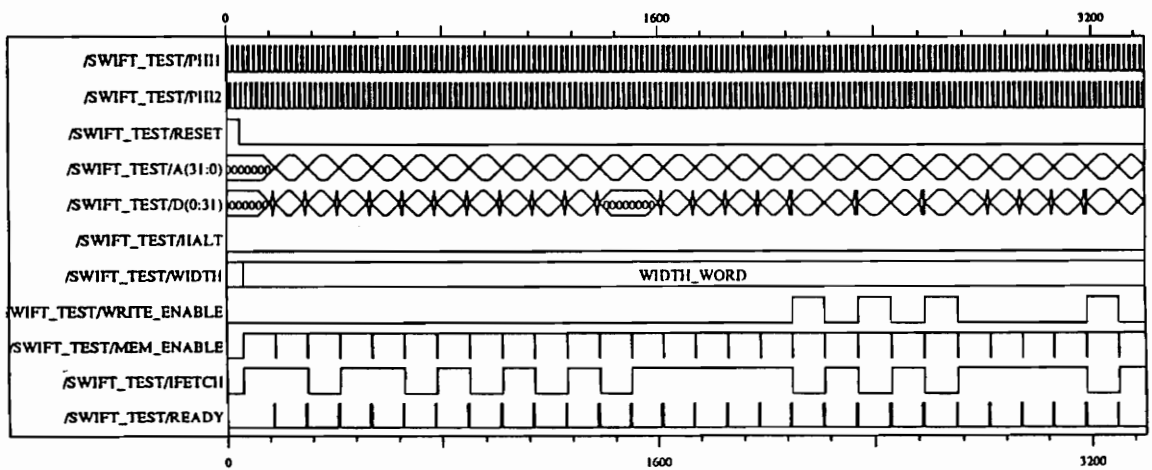
x:    .word 0,1,2
xtop: .word 3
a:    .word 3
```

Figure 3.9 shows the result obtained by executing the above code. It can be observed that three of the write\_enables' are closer to each other than in the earlier case, indicating that the results are stored consecutively, once they have been obtained. The simulation time is observed to reduce from 4000 ns to 3200 ns by eliminating overhead instructions through loop unrolling techniques.

Unfortunately, the parallelism embedded at the instruction level is rather limited. Very few processors have consistently executed more than two instructions per machine cycle over a wide range of



**Figure 3.8** Output result of a loop, used to add a scalar value to a vector in memory



**Figure 3.9** Output result of a loop after scheduling and unrolling operations

programs. Instruction level parallelism is often constrained by program behavior, compiler and operating system limitations, and execution mechanisms built into modern computer systems. A solution to this problem is to connect multiple processors together and distribute the program among the available processing nodes.

### **3.10 Conclusions**

This chapter examined design issues related to the VHDL model of the processing node, to be used in the multicomputer network described in the next chapter. The structural and behavioral level VHDL models of the processing node are based on the DLX processor, a 32-bit reduced instruction set processor. The DLX processor was chosen as the processing element to be included in the fat-tree multicomputer due to its features being very similar to those found in some of the commercially available reduced instruction set processors. The instruction set of the DLX processor was described, along with the instruction format used by the processor. Since parallel programs make efficient use of memory hierarchies, the design included a cache memory in each processing node.

The use of instruction level parallelism to decrease the cycles per instruction (CPI) was explored with a test-bench program. The fact that techniques used to increase parallelism in a single processor only increase the performance to a limited extent indicates that other approaches need to be considered in order to provide sufficient computing power in the future. One such approach, investigated in the following chapters, is the use of a multicomputer network with a large number of processing nodes connected together by an interconnection network.

## Chapter 4

### Model of the Fat-tree Interconnection Network

#### 4.1 Introduction

The von Neumann model of a computer is limited by the fact that all the communications between the central processing unit and the memory unit proceed along what is essentially a single interconnection line, the so-called "von Neumann bottleneck" [17]. The negative aspects of this single link on the speed of a computer are not confined only to its limited capacity. In fact, the entire algorithm design process is adversely affected since the programmer is forced to think about computational processes in sequential terms.

With advances made in digital component manufacturing, other structures have been considered (in Chapter 2) where multiple processors are coupled in different ways to produce many different interconnection network topologies. One such organization of multiple processing elements is the *binary tree*. This chapter begins with a series of definitions used to describe trees in graph theory. Examples are provided of earlier research projects which utilized tree-based structures. A major limitation of the planar tree structures is analyzed, along with a solution to make use of the *binary fat-tree* to help alleviate the problem. The *Universality theorem* is used to show that an N-node fat-tree (but not 2-D arrays or ordinary trees) can simulate any other N-node structure of the same volume in, at worst  $O(\log^3 N)$  time [15]. This fact is made use of in Chapter 5, when the fat-tree interconnection network is evaluated as a topology for the distributed simulation of VHDL models.



The VHDL model of the fat-tree with the inclusion of structural level processing nodes, differs from models created by others [8] in one important respect. VHDL models of multicomputer networks created previously were used mainly to model the routing among the interconnecting switches at the nodes of the network. The processing nodes were modeled at a behavioral level, without any processing capability. Inclusion of the structural level model of the processing node (described in Chapter 3) in the model of the fat-tree has a useful consequence. The instruction frequency mix of a given parallel program can be measured. Interestingly, this mix is seen to be different in the case of parallel programs than what would be found in a sequential program.

A practical aspect of efficiently simulating large behavioral and structural models, (using the fat-tree model as a case study) on a uniprocessor system is analyzed in Section 4.7. The system resources of the workstation are carefully monitored to see where resource utilization problems usually occur.

## 4.2 Tree Terminology

This section includes a set of definitions to help understand the terminology used when describing tree structures. In order to be concise, only a few definitions are provided. Numerous sources include further details [19, 20, 21].

**Definition 1.** A graph  $G$  with  $n$  nodes is called a *tree* if  $G$  is connected and has no cycles. An important property of trees is obtained by studying the relationship between the number of vertices and the number of edges of a tree: If  $T$  is a tree of  $n$  vertices and  $e$  edges, then  $n = e + 1$ .

Also, if  $G$  is any graph of  $e$  edges, and the degree of vertex  $u$  of graph  $G$  is  $\deg(u)$ , then the sum of all the neighbors of  $u$  is twice the number of edges, i.e.  $\sum \deg(u) = 2e$ .

**Definition 2.** A tree in which each node has no more than two *successors* or child nodes is called a *binary tree*. The nodes of a tree which have no child nodes are called *terminal nodes* or *leaves*. The subtrees of a binary tree are *ordered* in the sense that there is a *left* child and a *right* child.

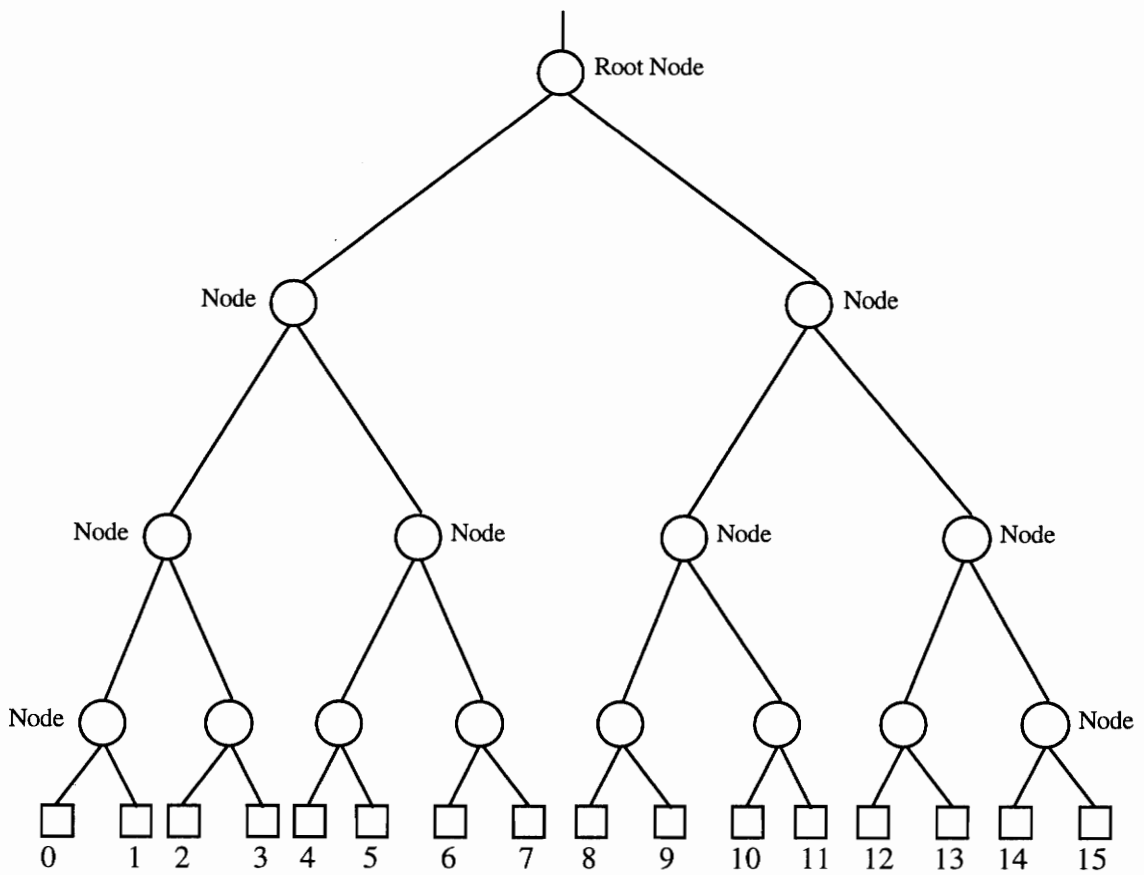
**Definition 3.** A binary tree is said to be *complete* if its non-terminal nodes all have exactly two successors.

**Definition 4.**  $T$  is a *balanced* binary tree if and only if for each node  $t$  in  $T$ , the depths of  $t$ 's right and left subtrees differ at most by one. If one subtree is null, the other subtree must be either null or a leaf.

### 4.3 Tree-based Structures

In general, a  $k$ -level, completely balanced binary tree has  $N = 2^k - 1$  nodes. The maximum node degree is 3 and the diameter is  $2(k-1)$ . With a constant node degree, the binary tree is considered to be a scalable architecture. Figure 4.1 shows a diagram of a 16-node complete binary tree. There have been research projects conducted at various universities, using binary tree-based structures. The Cellular Computer [22] constructed at the University of North Carolina was a tree machine designed specifically for parallel execution of the expressions found in programs written in "functional languages." The Non-von multicomputer [23] developed at Columbia University in the early '80s was designed to rapidly execute large-scale data-manipulation tasks. The DADO project [24], also developed at Columbia university, uses a binary tree-based machine as a special purpose computer to handle expert production problems.

All of these machines were developed with specific applications in mind. This is due to the fact that the binary tree's single root node can be a bottleneck when information is to be re-arranged rather than reduced, as in sorting data. When information is to be sent between non-adjacent pairs of nodes, a message going from processor  $i$  to processor  $j$  goes up the tree to their least common ancestor and then back down according to the least significant bits of  $j$ . Message traffic through lower-level nodes which are closer to the root is heavier than that of higher-level nodes. This leads to congestion in the lower-level channels. Augmented tree structures like the *binary fat-tree* and the *quadtree* have been used in commercial products to help reduce the channel bandwidth problem [25].



**Figure 4.1** A complete binary where all switches have one parent and two children. The processing nodes, labeled 0 to 15, are represented by the square boxes. ( $h=4$ ,  $c=2$ ,  $p=1$ )

#### 4.4 Universal Fat Trees

The parallel processing architectures being developed should be able to execute a variety of general-purpose parallel algorithms. The major drawback of special purpose architectures is the fact that they can typically compute specialized algorithms very quickly, but their performance diminishes considerably when general algorithms need to be computed. A fat-tree (FT) is a routing network based on a complete binary tree. A set  $P$  of  $n$  processors is located at the leaves of the fat-tree. Each edge of the underlying tree corresponds to a channel of the fat-tree. Each channel consists of a bundle of wires, and the number of wires in a channel  $c$  is called its capacity, denoted by  $c_k$ . The set of channels in FT is denoted by  $C$ , with  $c \in C$ . Each node is considered to have a level number that is its distance from the root and each channel  $c$  is considered to be at the same level as the node beneath it. If the channel at level 0 (the root) has capacity  $w$ , then FT is considered to have root capacity  $w$ .

**Definition 1.** Let FT be a fat-tree of  $n$  processors with root capacity  $w$  where  $n^{2/3} \leq w \leq n$ . Then if each channel  $c \in C$  at level  $k$  satisfies

$$c_k = \min \{ \lceil n/2^k \rceil, \lceil w/2^{2k/3} \rceil \}$$

FT is called a *Universal fat-tree*.

We can see that:

i) If  $k > 3 \log(n/w)$ , then  $\lceil n/2^k \rceil < \lceil w/2^{2k/3} \rceil$

Hence,  $c_k = \lceil n/2^k \rceil$

$$= (n+1)/2^k$$

$$= 1, 2, 4, 8, \dots, \text{ for } k = \log(n+1), \log(n+1) - 1, \log(n+1) - 2, \quad (1)$$

ii) If  $k \leq 3 \log(n/w)$ , then  $\lceil n/2^k \rceil \geq \lceil w/2^{2k/3} \rceil$

Hence,  $c_k = \lceil w/2^{2k/3} \rceil$

$$= \dots, w/8^{2/3}, w/4^{2/3}, w/2^{2/3}, \text{ for } k = \dots, 3, 2, 1. \quad (2)$$

It can be seen from the above result (1), that the channel capacities of a Universal fat-tree grow exponentially as we go up the tree from the leaves [2]. Initially, the capacities double from one level to the next, but at levels less than  $3 \log(n/w)$  away from the root, the channel capacities grow at the rate of  $4^{2/3}$ . Thus, the problem of congestion seen near the root of simple tree-based structures is alleviated with the increased channel capacity of the fat-tree network.

A *Universal* circuit of a given size can be programmed to simulate any circuit whose size is only slightly smaller [15]. In other words, for a given amount of communications hardware, a fat-tree built from that amount of hardware can simulate any other network built from the same amount of hardware, using only slightly more time (a polylogarithmic factor greater).

Routing networks for parallel processing machines are usually analyzed in terms of performance and cost. Performance is typically measured by the amount of time taken to route an arbitrary set of messages. Cost is usually measured by the number of switching components and wires used. Cost can also be measured as the "volume" of the physical implementation of the network [15].

The Universality theorem proves that an  $N$ -node fat-tree can simulate any other  $N$ -node structure of the same volume in, at worst  $O(\log^3 N)$  time. Polylogarithmic time in parallel computation corresponds to polynomial time in sequential computation. Networks which are not universal (for example, 2-D arrays, simple trees or multigrids) exhibit polynomial slowdown when simulating other networks. Thus, they have no theoretical advantage over a sequential computer which can easily simulate a network with polynomial slowdown. The theorem follows from the fact that a graph can be successively cut and decomposed, thus giving a binary tree. Details of the Universality theorem are provided in [15].

#### 4.5. VHDL model of the Fat Tree

In a fat-tree, the processors are located at the leaves of a complete binary tree, while the nodes comprise the *switches*, which are used to route information through the network. The higher a switch is placed in the tree structure, the more connections that are routed through it. The VHDL description of the

fat-tree model consists of a number of instances of a single basic component, the switch, interconnected in a regular pattern. In VHDL, the *generate* statement can be used to repeatedly instantiate the switch and its connections. The tree structure has been implemented using a "recursive" structure [26]. A recursive structure is one which is parametrized with respect to its size, and which is described in terms of smaller instances of the same structure. The tree, with its root cell at height  $h$ , is connected to a number of subtrees of height  $h-1$ . The entity declaration shown below describes the interface of the tree.

```
entity fat_tree is
  generic (h, c, p : positive)
  port ( receive_msg_from_child: in connection_vector(0 to c**h-1);
        receive_ack_from_child: in connection_vector(0 to c**h-1);
        receive_msg_from_parent: in connection_vector(0 to c**h-1);
        receive_ack_from_parent: in connection_vector(0 to c**h-1);
        send_msg_to_child:      out acknowledge_vector(0 to p**h-1);
        send_ack_to_child:      out acknowledge_vector(0 to p**h-1);
        send_msg_to_parent:      out acknowledge_vector(0 to p**h-1);
        send_ack_to_parent:      out acknowledge_vector(0 to p**h-1));
end fat_tree;
```

The *fat\_tree* of height  $h$  has  $c$  child connections for communication with child switches and  $p$  parent connections to communicate with parent switches in the interconnection network. The type *connection\_vector* is used to represent a collection of message connections between switches. The type *acknowledge\_vector* is used to synchronize message-passing.

An outline of the architecture body of the fat-tree is shown below:

```
architecture recursive of fat_tree is
  component switch
    generic (...)
    port (...)
  end component;

  component fat_tree
    generic (h: positive,...)
    port (...)
  end component;

begin
  repetition_of_switch: for i in ... generate
    the_switch: switch
```

```

    generic map (...)
    port map (...)
end generate repetition_of_switch;

repetition_of_fat_tree: for i in ... generate
    recursive_instance_of_tree: fat_tree
        generic map (h-1,...)
        port map (...)
    end generate repetition_of_fat_tree;
end recursive;

```

The component *switch* represents the basic component from which the fat-tree is constructed. Each switch has *c* child connections and *p* parent connections. It can route messages between its children directly and can forward messages for *nephews* via any of its parents. Messages are used to communicate information between the various processing nodes. The message header contains information used for routing, including the addresses of the source and destination nodes. In addition, a record of the path taken through the network and the number of switch elements (hops) through which the message passed is also included in the message header. This information is required by the machine description module, described in Chapter 5. Figure 4.2 shows the path taken by the messages as they travel from processor 0 to all the other processors in a 16-node binary fat tree network.

```

Message 0 from 0 to 0 via 1 hop: (1,0,0)
Message 1 from 0 to 1 via 1 hop: (1,0,0)
Message 2 from 0 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Message 3 from 0 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Message 4 from 0 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Message 5 from 0 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Message 6 from 0 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Message 7 from 0 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
Message 8 from 0 to 8 via 7 hops: (1,0,0) (2,0,0) (3,0,0) (4,0,0) (3,0,1) (2,0,2) (1,0,4)
Message 9 from 0 to 9 via 7 hops: (1,0,0) (2,1,0) (3,2,0) (4,4,0) (3,2,1) (2,1,2) (1,0,4)
Message 10 from 0 to 10 via 7 hops: (1,0,0) (2,0,0) (3,1,0) (4,2,0) (3,1,1) (2,0,2) (1,0,5)
Message 11 from 0 to 11 via 7 hops: (1,0,0) (2,1,0) (3,3,0) (4,6,0) (3,3,1) (2,1,2) (1,0,5)
Message 12 from 0 to 12 via 7 hops: (1,0,0) (2,0,0) (3,0,0) (4,1,0) (3,0,1) (2,0,3) (1,0,6)
Message 13 from 0 to 13 via 7 hops: (1,0,0) (2,1,0) (3,2,0) (4,5,0) (3,2,1) (2,1,3) (1,0,6)
Message 14 from 0 to 14 via 7 hops: (1,0,0) (2,0,0) (3,1,0) (4,3,0) (3,1,1) (2,0,3) (1,0,7)
Message 15 from 0 to 15 via 7 hops: (1,0,0) (2,1,0) (3,3,0) (4,7,0) (3,3,1) (2,1,3) (1,0,7)

```

**Figure 4.2.** Hop count of the messages, and message path for a 16-node fat tree

Each time a switch forwards a message from one node to the next, it waits for an acknowledgement from the node it has forwarded the message to. The switch then sends an acknowledgement to the child or parent from which it received the message. The co-ordinates of each switching node through which a message passes from source to destination is provided by the  $(X,Y,Z)$  indices in Figure 4.2.  $X$  is the vertical level number of the bottom row of switches, counting from 1 at the bottom.  $Y$  is the back-to-front horizontal index of the top column of switches and  $Z$  is the left-to-right index of the top column of switches. The routing algorithm for tree-based networks is straight-forward. A message travels from the source node until it reaches the least common ancestor of the destination node while ascending up the tree. From this point, the message is forwarded down the tree to its destination. At each node on the way down, a determination is made about whether the message should be forwarded to the left or right child node.

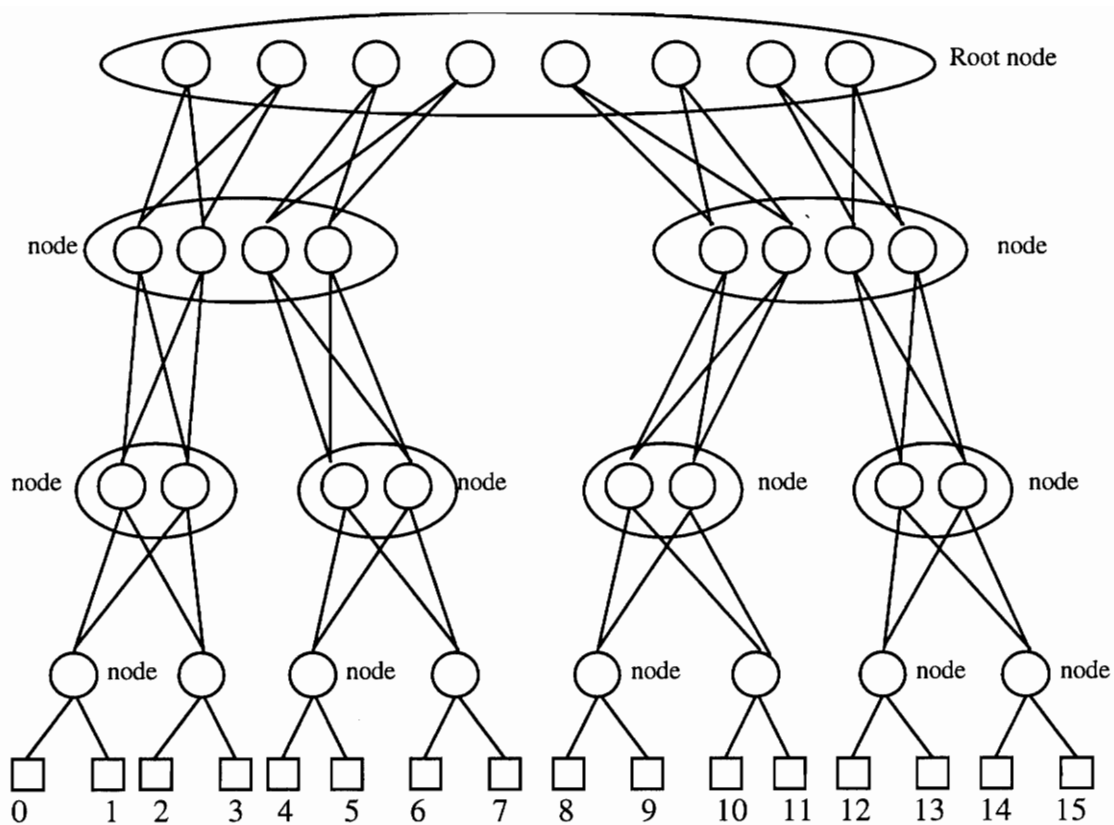
Figure 4.3 shows a diagram of a 16-node binary fat-tree. All the non-root nodes have two parents. The parameters used are  $h = 4$ ,  $c = 2$  and  $p = 2$ . If the ovals are viewed as nodes and the edges between a pair of nodes as a single edge, a tree structure can be seen. In a quadtree, each node has four children. The commercial implementation of the CM-5 uses a quadtree interconnection network [25]. The processing elements are located at the leaves, with the internal nodes comprising switches.

Figure 4.4 shows a diagram of a 16 node quad tree with each switch having two parents and four children. The parameters used are  $h = 2$ ,  $c = 4$  and  $p = 2$ .

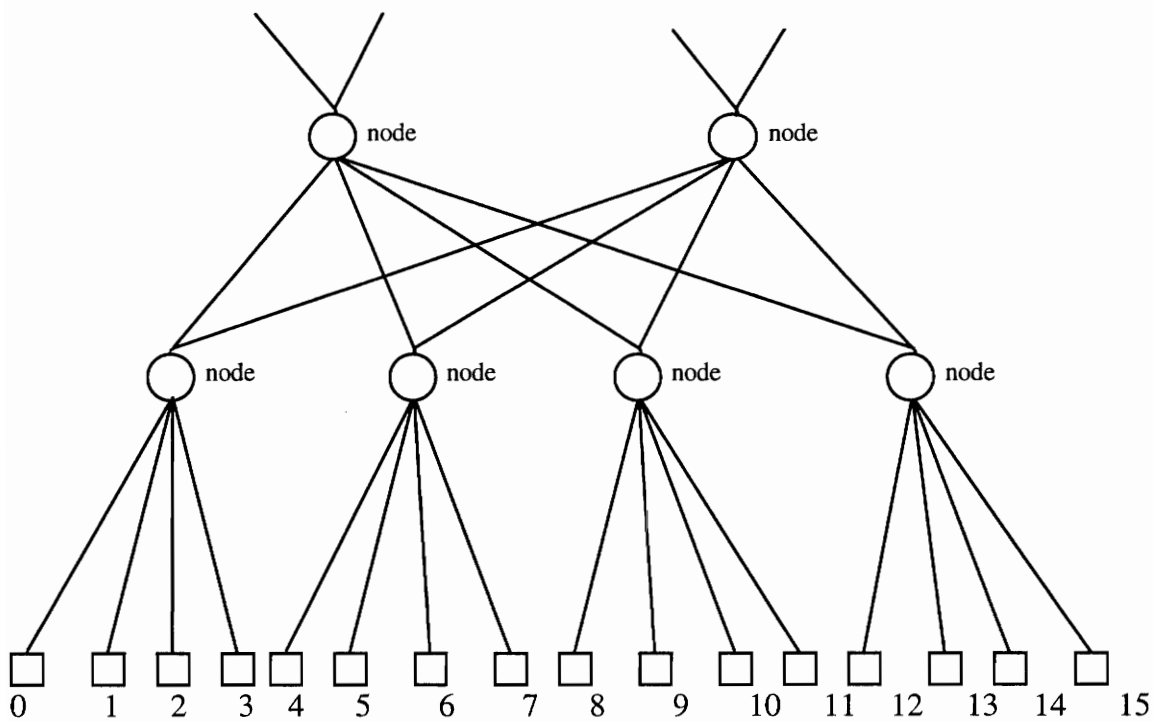
## 4.6 Dynamic Instruction Frequency Counts

Two aspects of the processing nodes used in the fat-tree are analyzed in this section. First, the efficient utilization of the memory hierarchy in parallel programs run over multicomputers, where each processing node utilizes its private cache and memory, is seen to provide a significant cost benefit when compared to vector supercomputers which usually do not use cache memories. Secondly, dynamic instruction mix statistics of benchmark programs run on actual implementations of multicomputers like





**Figure 4.3** A 16-node binary fat-tree, where all non-root switches have two parents. If the ovals are viewed as nodes and the edges between a pair of nodes as a single edge, a tree structure can be seen. The processing nodes labeled 0 to 15 are represented by the square boxes. ( $h = 4$ ,  $c = 2$ ,  $p = 2$ )



**Figure 4.4** A 16-node quadtree, where all non-root switches have two parents, and four children. The processing nodes labeled 0 to 15 are represented by the square boxes. ( $h=2$ ,  $c=4$ ,  $p=2$ ).

the CM-5 [27], are seen to differ from the instruction mix observed in sequential uniprocessors. Having the ability to measure the instruction mix statistics in programs run using the VHDL model of the fat-tree is therefore seen to be of value to the computer systems architect.

An interesting observation can be made about dynamic instruction mix statistics for benchmark programs which were run on a 128-node CM-5 machine [27]. The instruction mix is shown in Table 4.1, for the largest sample programs, ranging in size from 50 to 1100 lines. *ALU* includes integer and floating point arithmetic, *messages* include instructions executed to handle messages, *heap* includes global I-structure and M-structure accesses and *control* represents all control-flow operations instructions.

Program	Control	Heap	Messages	ALU
QS	60.3 %	6.2 %	19.5 %	14 %
Gamteb	58.1 %	6.3 %	19.6 %	16 %
Paraffins	54.8 %	15.6 %	14.4 %	15.2 %

**Table 4.1** Dynamic instruction mix statistics of parallel benchmark programs [27]

*QS* is a simple quick-sort program using accumulation lists [27]. *Gamteb* is a monte Carlo neutron transport program, which is highly recursive with many conditionals. *Paraffins* is a program used to enumerate the distinct isomer of paraffins. Notice that the programs are control intensive and make frequent remote references.

On the other hand, dynamic instruction mix statistics obtained from sample benchmark programs run on sequential uniprocessor machines are less control intensive with a much higher percentage of arithmetic operations. Table 4.2 shows the dynamic instruction mix statistics of benchmark programs run on MIPS R2000 processor. *GCC* is a C compiler program, *TeX* is a document-processing program and *US Steel* is a widely used COBOL benchmark program [1].

Program	Control	Data Transfer	ALU	Other
GCC	24%	27%	35%	14%
TeX	10%	33%	41%	16%
US Steel	23%	10%	49%	18%

**Table 4.2** Dynamic instruction mix statistics of benchmark programs run on a uniprocessor [1]

The percentage of ALU operations seen in Table 4.2 for the benchmark programs is observed to be much higher as compared to the programs in Table 4.1. Different instruction mix statistics are observed in the case of sample programs run on sequential machines as compared to highly parallel programs run on parallel processing systems. However, it must be noted that the results obtained from the measurements have not been proven to arise from any inherent property of computation, but rather from extensive observations taken from traces on actual machines. Therefore, it would be useful for the computer systems designer to be able to measure the instruction mixes of sample programs when building prototype machines. The VHDL model of the fat-tree network can be used to monitor the various operations being performed during a simulation run. A list produced after the simulation run can then be analyzed to determine the instruction mix statistics of a sample program. This is a novel approach to using the VHDL language to measure instruction mix statistics of sample benchmark parallel programs, which are usually run on multicomputer networks.

A sample output is shown in Table 4.3 of the dynamic instruction frequency count of operations while running a parallel program on the VHDL model of the fat-tree. An example of a sequential program that is easily parallelizable and run simultaneously among multiple processors is given in Appendix A. The instruction mix of the testbench program shown below has similar characteristics to the parallel programs in Table 4.1 with relatively few ALU operations and a greater number of control operations.

Instruction	Freq.	Instruction	Freq.	Instruction	Freq.	Instruction	Freq.
ADD	5	ADDI	1	ADDUI	0	ADDU	0
AND	0	ANDI	0	BEQZ	23	BNEZ	5
J	0	JAL	7	JALR	0	JR	1
LB	0	LBU	0	LW	29	LF	0
SD	28	SEQ	0	SW	0	SF	0

**Table 4.3** Dynamic instruction frequency counts obtained from the execution of a test program on the VHDL model of the 32-node fat-tree multicomputer.

## 4.7 System Resource Utilization

In analyzing the computer systems resource utilization, it was observed that the workstation (Sun Sparkstation 2) used to run the VHDL simulations had resource utilization problems. The use of *recursive* subprograms should be avoided if CPU time performance is an problem. Recursive codes, because of their stacked contexts, are more expensive than the iterative form [28]. However, when a recursive hardware structure is to be described in VHDL, it has been observed that the recursive structures are much easier to develop than equivalent repetitive structures. The main difficulty in developing the repetitive structure lies in devising a way of interconnecting the basic components comprising the structure [26].

### 4.7.1. Monitoring System Resources

In any computer system, there are three fundamental resources: the CPU, the memory and the I/O system. CPU utilization can be monitored to determine if there is CPU *contention* occurring during a program run. CPU contention is often seen in multi-user, time-sharing systems when many programs want to use the CPU at the same time. At some point, the number of CPU cycles needed to be executed by the various programs is more than the CPU can allocate and this leads to contention. However, it is important to make sure that the problem occurs due to the lack of CPU processing power. For example, a shortage of memory or a slow I/O can cause the system performance to degrade drastically, but the CPU

might be spending most of its time in an idle state. In this case, using a faster CPU would only increase the time it spends being idle [29].

Memory contention arises when the memory requirements of the active processes exceed the physical memory available in the system. To handle this lack of memory without crashing the system or killing processes, the system starts *paging*, i.e., moving portions of active processes to disk in order to reclaim physical memory. Paging is distinguished from *swapping*, which means moving entire processes to disk and reclaiming their space. Paging and swapping indicate that the system cannot provide enough memory for the processes that are currently running.

The I/O subsystem is a common source of resource contention problems. A finite amount of I/O bandwidth must be shared by all the programs that are being run at any given point in time. A problem arises when more data needs to be transferred across the system's I/O buses than the I/O buses are able to provide.

One of the important issues in simulating large and complex models is the time that the simulation "actually" takes to run, known as *run time*. In order to quantitatively measure the effect of VHDL model size on the run time, a series of tests were conducted. The workstation used for the simulations was a Sun Sparkstation 2 running SunOS 4.1.3. Each processor of the fat-tree was programmed to run code of low algorithmic complexity, involving only a few hundred arithmetic operations. If the algorithmic complexity of the program was increased by an order of magnitude, the run time of the simulation also increased as a result of the greater number of calculations to be executed. The model of the fat-tree was increased in size and the run time measured. Table 4.4 reports on the memory paging activity for a 16-node fat-tree model and Table 4.5 reports on the memory paging activity for a 32-node fat-tree model. It can be clearly seen from the *pi* and *po* columns that there is a great deal of paging activity taking place when the 32-node model is used to perform the simulation, while the simulation of the 16-node model does not suffer from a shortage of available memory to the same extent. The *pi* column indicates the number of pages per second that have been moved from disk to memory. The *po* indicates the number of

**Table 4.4** A report on memory paging and swapping information generated using the *vmstat* command, for a 16-node fat-tree simulation on a Sun Sparksation 2.

Time	Memory Paging					Disk Faults						CPU			
(min.)	pi	po	fr	de	sr	s0	s1	s2	d3	in	sy	cs	us	sy	id
0	0	0	0	0	0	0	0	0	0	10	28	11	0	1	0
3	0	0	8	0	13	1	0	0	0	31	436	71	92	8	0
6	0	0	4	0	9	0	0	2	0	22	356	58	95	5	0
9	0	0	0	0	7	1	0	0	0	17	353	57	92	8	0
12	0	0	28	16	36	1	0	0	0	13	471	82	90	10	0
15	0	0	16	0	24	10	0	0	0	13	421	71	89	11	0

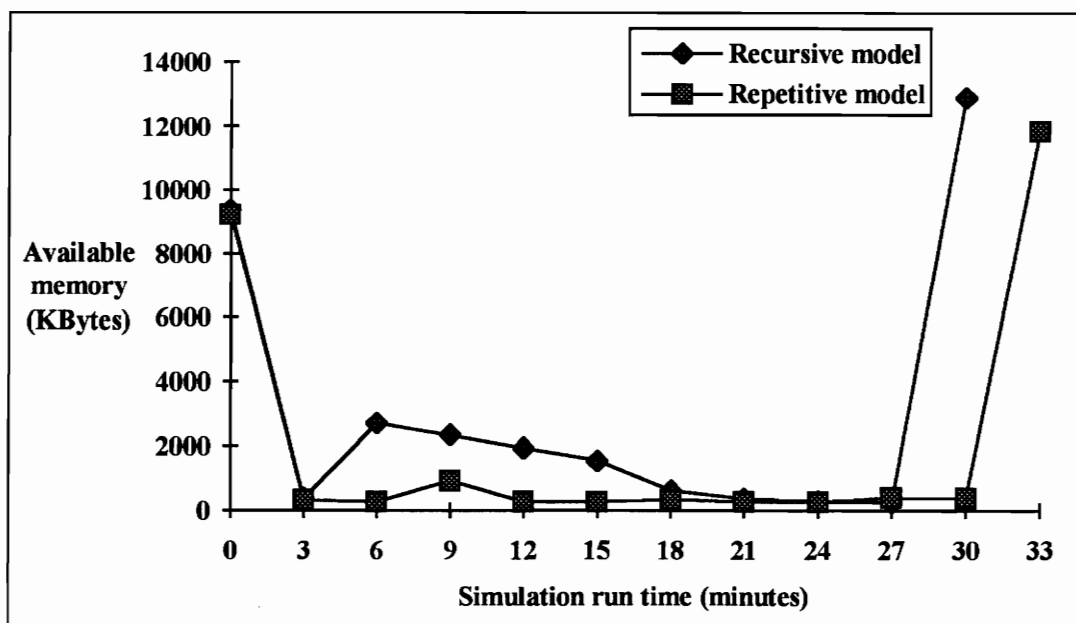
**Table 4.5** A report on memory paging and swapping information generated using the *vmstat* command, for a 32-node fat-tree simulation on a Sun Sparksation 2.

Time	Memory Paging					Disk Faults						CPU			
(min.)	pi	po	fr	de	sr	s0	s1	s2	d3	in	sy	cs	us	sy	id
0	112	88	96	48	14	31	0	0	0	129	229	69	80	20	0
3	120	52	60	0	13	3	0	0	0	161	228	88	91	6	2
6	76	32	36	0	8	17	0	2	0	116	217	77	87	13	0
9	60	16	20	0	4	1	0	0	0	104	241	72	96	4	0
12	36	8	12	0	2	0	0	0	0	70	230	57	97	3	0
15	20	0	4	0	0	2	0	0	0	49	216	49	93	5	2

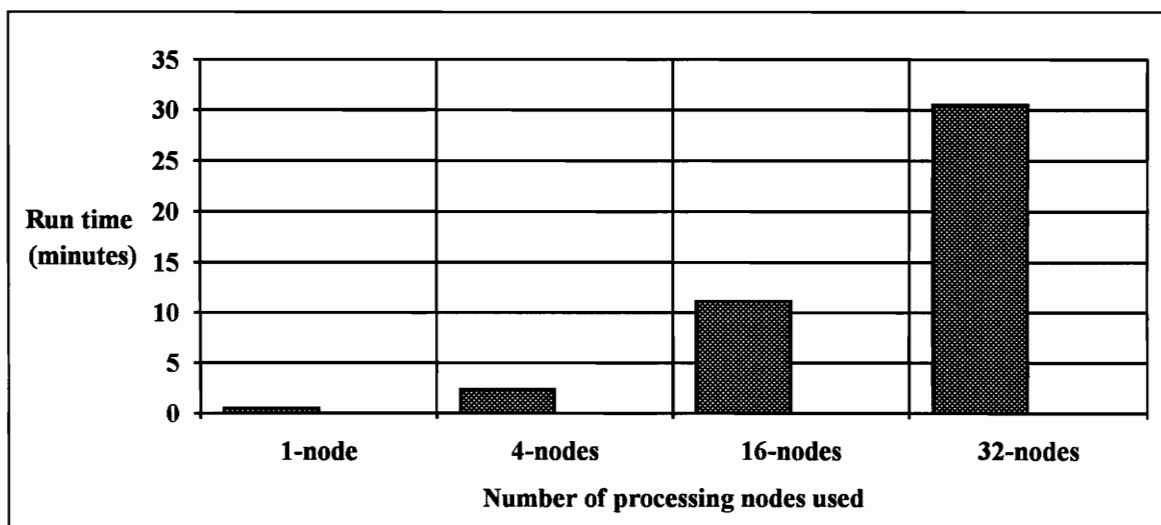
pages per second that have been moved from memory to disk. There are two columns labeled *sy*. The rightmost column reports the percentage of total CPU time spent in the system state. The *us* column indicates the percentage of total CPU time spent in the user state. The simulation was found to be very CPU intensive and the CPU was rarely idle as can be seen in the *id* column. In smaller models of the fat-tree, there was a negligible amount of paging observed and the performance did not suffer to the same extent. In order to gain a clearer insight into the factors affecting the run time of the simulation, a comparison was made of the available memory resources when simulating a 32-node fat-tree with a recursive structure (used to instantiate the connections between the switches) and with a non-recursive (repetitive) structure. Figure 4.5 shows a comparison of the available memory at different durations in time during the simulation run. The run time of the two simulations did not vary by a significant amount. Thus the load on the workstation used for the simulation in terms of CPU utilization was the dominating factor in determining the run time of the simulation of large VHDL models on a uniprocessor with memory contention caused by the size of the model, adding to the problem.

It was observed that the run time increased greater-than-linearly with an increase in the number of processing nodes since the CPU utilization of the workstation was increased. Figure 4.6 shows that it took 2.3 minutes to simulate a 4-node fat-tree network. To simulate a 16-node network, running the same program, the run time increased to 11.1 minutes. When the number of processing nodes was increased to 32, the run time nearly tripled to 30.5 minutes. When the memory paging activity was observed to significantly increase for increasing numbers of processing nodes in the fat-tree model, there was a  $O(n)$  growth in run time seen in the simulations. This indicates that once memory contention problems started to occur, the performance of the simulation on the uniprocessor workstation degraded to a considerable extent. Since the number of processing nodes in most multicomputer networks increases by a factor of 2 (when the size of the network is increased), a growth rate of  $O(n)$  in simulation run time quickly becomes untenable when multicomputers containing thousands of processing nodes need to be simulated. A





**Figure 4.5** Comparison of the available memory resources for a 32-node fat-tree when a recursive structure is used and when a non-recursive (repetitive) structure is used.



**Figure 4.6** The run time is seen to increase in a non-linear way, due to memory contention, with an increase in the size of the fat-tree model.

solution to this problem is to use parallel techniques to simulate large-scale system-level VHDL models. Various distributed simulation approaches are considered in the next chapter.

## **4.8 Conclusions**

This chapter presented research dealing with aspects of effectively modeling multicomputers using the VHDL language. Multicomputer networks based on tree structures have been constructed at various research organizations. Their success has been limited when it came to using them for a wide variety of applications and they were used mainly to solve a few special-purpose algorithms. This was due to the fact that when many messages had to pass through the root-node, the channel through which the information would have to pass would quickly become congested.

The binary fat-tree architecture was seen to be able to alleviate the root congestion problem by increasing the channel capacity nearer the root. Also, the fat-tree is a Universal structure, in the sense that it can simulate any other network with only a polylogarithmic loss in time. VHDL models of multicomputers developed previously were limited in the sense that they had very simple models of processing nodes and hence their use was limited. The fat-tree model can be used to run sample programs and the dynamic execution of instructions can be recorded.

Large VHDL models present a problem since they are computationally intensive and they utilize a large portion of the systems available memory. Successively larger models were created and the system resources monitored during simulation runs to determine where "bottlenecks" usually occur. The chapter thus dealt with the efficient simulation of large VHDL models on a uniprocessor. The next chapter will report on research conducted to partition the VHDL code amongst a set of processors, and execute the code in parallel. The fat-tree architecture is proposed for the parallel discrete event simulation of VHDL models and a comparative analysis made, to evaluate the feasibility of this approach.

## Chapter 5

### Parallel Simulation of VHDL models over a Fat-tree Interconnection Network

#### 5.1 Introduction

One of the problems observed in simulating complex models is the fact that simulation runs take a very long time to execute. The trend is for run times to lengthen, since the combinatorial complexity of circuits is out-stripping performance improvements in computers. In the previous chapter, it was observed that the increased run times for the execution of VHDL models became an important factor in the overall design process, when the size and complexity of the model was increased.

In this chapter, a multicomputer using the fat-tree interconnection network is proposed as a suitable architecture for the parallel simulation of VHDL models. Various algorithms used for the parallel discrete event simulation (PDES) of VHDL models are evaluated. The feasibility of this approach is carried out with the help of a "Parallelism Analyzer," developed at the University of Genoa, Italy.

Results obtained from simulating the model of the fat-tree multicomputer network, described in Chapter 4, are used to provide details to the analyzer about the interconnecting links between the various processing nodes. The number of hops a message takes to travel from one processor to another in the fat-tree is used by the Parallelism Analyzer to estimate the time of an event message between two processors. The *roll-back* costs and the cost of communication amongst the processing nodes are taken into consideration when evaluating the *speedup* of the simulation time of a VHDL model, simulated over multiple processors. The speedup of the simulation of a VHDL model using the fat-tree topology is compared with the results obtained with a linear array topology. In conclusion, the future inclusion of the

"shared variable" into the language and its impact on the implementation of parallel simulators on multicomputer networks are analyzed.

## 5.2 VHDL Parallel Discrete Event Simulation

A discrete event simulation model assumes the system being simulated only changes state at discrete points in simulation time. The behavior of a microprocessor, with a clock causing events to occur at periodic times, is an example of a system suitable for discrete event simulation. A simulation model consists of a number of "logical" processes, each corresponding to a physical process, which communicate by exchanging time-stamped *event* messages. An event on a signal is defined by its new value and by the time when the new value will be assigned, also known as a "time-stamp." Each logical process contains sequential code, which is triggered by events scheduled for it by other logical processes. Correct execution of a distributed simulation model can be guaranteed if the following causality constraint can be satisfied: that a process receives and acts upon events in non-decreasing time-stamp order, unless the events are independent of one another. Two events are considered to be independent of one another if processing one, has no effect on the outcome of processing the other. Thus, independent events can be processed in any order, irrespective of their time-stamps. A number of approaches are used in distributed simulation, such as *centralized-queue* PDES and *distributed-time* PDES algorithms. Distributed PDES algorithms are further divided into *conservative* and *optimistic* algorithms. Each of these algorithms have different approaches to dealing with parallelism, synchronization and communication of data within the model. In a central-queue PDES simulator, events are sent to a logical process by the kernel process. In a distributed PDES simulator, events are sent from one logical process to another via event message links. A sequential DES approach is initially discussed, followed by various PDES approaches.

### 5.2.1 Sequential Discrete Event Simulation

A typical sequential implementation of DES uses a time ordered central queue of events remaining to be simulated at later simulation times. The logical processes are implemented as co-routines, being

resumed in turn as the simulation progresses. Conceptually, the local clocks of all the logical processes are subsumed by a global simulation time clock. All the logical processes are initialized before the first simulation cycle. In each cycle, an event at a single instance in simulation time is processed. Since the queue is ordered by the time stamp of the events, the events are processed in non-decreasing time-stamp order, thereby satisfying the causality constraint. The order of processing of events scheduled for the same simulation time is not specified since such events correspond to simultaneous events in the physical system being modeled.

### 5.2.2 Centralized-queue Algorithms

A simple way to achieve parallelism in executing a simulation model is to execute all events selected in one time step, in parallel. Since these events all have the same time stamp, the causality constraint is not violated. If many events occur with exactly the same time-stamp, this method may give us significant speedup. However, in real models which include propagation delays, the probability of events being spread more evenly through simulation time is high. Hence the parallel software processes which receive these events will not be executed in parallel, even if the events are independent. Thus, the use of a global time clock and the need for synchronized time advances limit the amount of concurrency that can be exploited. This approach works efficiently with multiprocessors with only 10 nodes or so (for example, Zycads ViP multiprocessor architecture), but so far it has not been found to be effective for larger machines [30].

### 5.2.3 Distributed-time Algorithms

In the distributed approach, events are executed in parallel, regardless of their time-stamps. Each process can proceed with its own "local time." The processes communicate with each other using time-stamped messages. There are two basic algorithms that can be applied to VHDL simulation: conservative and optimistic distributed-time algorithms [31].

#### 5.2.3.1 Conservative Distributed-time Algorithms

In the conservative distributed-time approach, information about scheduled events is distributed amongst the receiving processes. Different processes may respond in parallel to events occurring at

different simulation times, but a process receives an event message only when it can determine that no causality error will occur [32].

Each logical process in the simulation maintains its own local clock, recording the simulation time it has reached. The conservative algorithm must ensure that a process does not receive an event with a time-stamp earlier than the current local clock. Events are sent between logical processes via "links" which are created at the beginning of the simulation. Each link has a clock associated with it, recording the time-stamp of the earliest message queued in the link. If the link is empty, a link clock records the time-stamp of the last message sent over the link. Each process repeatedly selects a link with earliest clock, and processes the first event message from the link. One of the major drawbacks with conservative algorithms is the fact that they may not take full advantage of all the "potential parallelism" present in the model. If there is any possibility that one process may effect another, then the process executions are performed sequentially.

#### 5.2.3.2 Optimistic Distributed-time Algorithms

Optimistic algorithms allow execution to proceed without regard to causality violations and take corrective action when violations are detected. In contrast to conservative algorithms, optimistic algorithms, using the *Time Warp* paradigm [32], do not need to determine when it is safe for execution to proceed. An advantage of the optimistic approach is that the parallel simulator can exploit parallelism in situations where causality errors might occur, but in fact do not.

Each process includes a local clock which measures simulation time for that process, and a queue of time-stamped event messages to be processed. Each process proceeds in simulation time at its own pace, on the assumption that no causality errors result. If at some point in simulation time, an event arrives with time-stamp less than the local clock (a *straggler* event), then the receiving process must *roll back*. It must undo all state changes back to a simulation time at or before the time-stamp of the straggler, and only then can it recommence. When roll-back occurs, the process must send an "anti-message" for each message sent with time-stamp greater than the straggler event. Optimistic algorithms rely on the

frequency of rollback being sufficiently low that it does not significantly affect the progress of the simulation.

### 5.3 The VHDL Parallelism Analyzer

In the previous section, various approaches commonly used for parallel discrete event simulations were discussed. The optimistic distributed-time algorithm is considered to be the most promising approach for parallel VHDL simulation over a multicomputer network [30]. This is primarily due to the optimistic algorithms allowing process execution to proceed without regard to causality violations, with corrective action being taken only when violations are detected.

In this section, a technique used to evaluate the performance of optimistic time algorithms on a fat-tree multicomputer network is described. Two simulation methods commonly used to study the performance of parallel computer architectures are *execution-driven simulations* and *trace-driven simulations*. Execution driven simulations require the use of complex compilers, with the distributed algorithms embedded within the run-time kernel. Trace-driven simulations require a time-ordered record of events, obtained from simulation runs on the system to be analyzed. The trace-driven approach has been used by the Parallelism Analyzer as it requires a much smaller development effort and is easier to implement [34].

The Parallelism Analyzer (PA) is used to emulate the behavior of a distributed simulation, by analyzing the events of a sequential simulation and mapping the sequential events to the corresponding processors, while ensuring that the dependency constraints are satisfied. Three inputs are required by Parallelism Analyzer: the source code of a VHDL benchmark program, the instruction trace from a sequential simulator, and information about the interconnection network to be evaluated.

The PA analyzes the VHDL source code and extracts "process dependencies." A VHDL process is a concurrent statement containing a series of sequential statements. During simulation, the process executes until a wait statement is reached, and is considered to be a single concurrent action. The

Parallelism Analyzer considers a VHDL process to be a basic unit which cannot be subdivided amongst multiple processors since a process contains serial code.

After the process dependencies in a VHDL program have been determined, the PA requires information about the fat-tree interconnection network in order for the processes to be allocated to the available processors. Results obtained from simulating the VHDL model of the fat-tree (Figure 4.2), in the form of the number of processors used, the minimum distance or "hop count" between pairs of processing nodes (in terms of the minimum number of nodes a message has to traverse), and the links connecting the various processing nodes, are used by the Parallelism Analyzer to estimate the transmission time of an event message between two processors.

The Parallelism Analyzer distributes the VHDL processes amongst the processing nodes of the fat-tree interconnection network. When the number of VHDL processes is greater than the number of available processors, the processes need to be efficiently distributed in order to minimize the communication between the processing nodes. The Min-Cut algorithm [35] is used to minimize the number of cuts (inter-processor communication) between two subsets (processors), while evenly balancing the "load" on each subset. The load is an estimate of the execution time of a VHDL process, derived from the number of statements comprising the process and the number of signal assignments present in the process.

## **5.4 Experimental Results and Analysis**

In order to analyze the feasibility of performing VHDL parallel discrete event simulations on a fat-tree interconnection network, three test programs were investigated and a number of tests were performed using these programs. The programs used are part of a suite of testbench programs [34], used to evaluate the performance of distributed simulation systems at the behavioral and structural level. They are representative of the kind of programs initially modeled at a higher level of abstraction before the models are synthesized. The size of the VHDL programs is seen to vary from 493 lines in one program to 4396

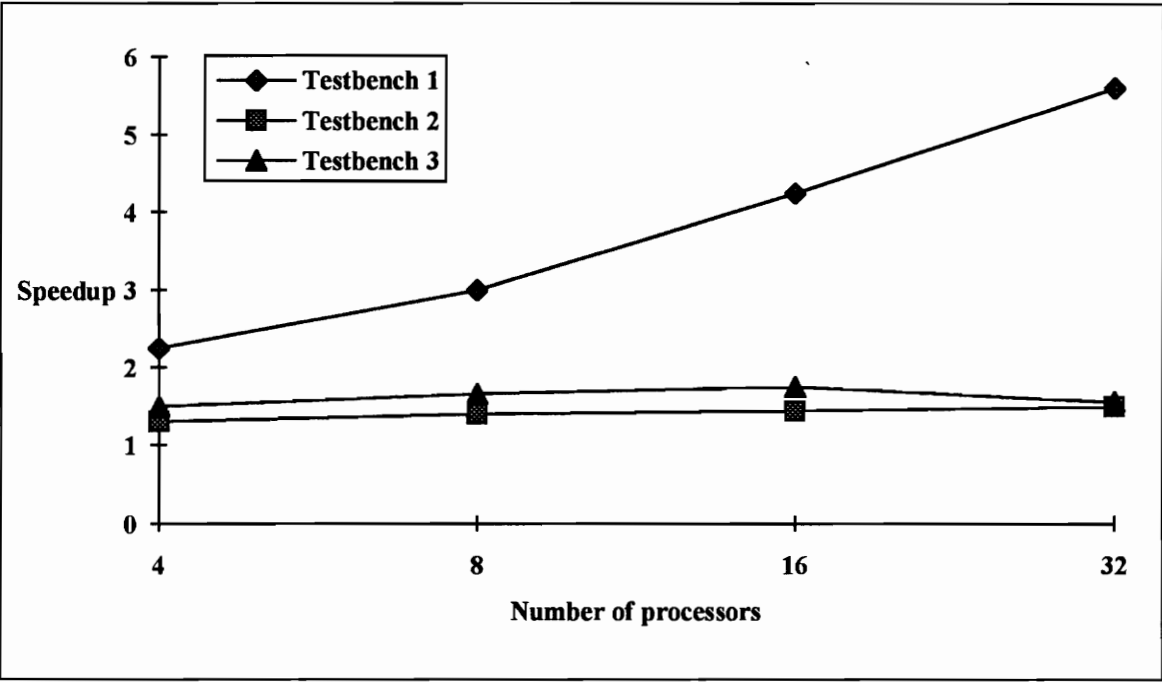


lines in another testbench program. The grain size of the programs, corresponding to the number of instructions per simulation cycle varies from 8 in one program to 141 in another program. Testbench 1 is a behavioral model of a butterfly processor, used for the computation of butterfly operations on complex data. Testbench 2 is a structural model of parallel-serial encoders. Testbench 3 is a behavioral model of a pipelined image processor, operating a mask convolution of a 2D image [34]. Table 5.1 provides static and dynamic information on the benchmark programs used in the tests.

Programs	VHDL lines	Number of simulation cycles	Number of events	Instructions/ cycle	Instructions /event
Testbench 1	493	280	3042	141	15
Testbench 2	351	28,776	34,671	8	6
Testbench 3	4,396	79,835	297,729	18	5

**Table 5.1** Static and dynamic information on the VHDL models used in the tests

The first simulation experiment considers the use of a centralized event queue algorithm for the parallel discrete event simulation of VHDL models on the fat-tree architecture. Table 5.2 shows the speedups obtained when the number of processors was increased from 4 to 32. Figure 5.1 displays a plot of the speedup observed with increasing numbers of processors. The speedup is defined as the ratio of the execution time of a VHDL simulation executed on a uniprocessor, to the execution time of the same simulation executed in parallel, on multiple processors. It is observed that the speedup is limited to 5, even when 32 processors are used. Thus, it is not feasible to use this approach on distributed memory architectures as the performance improvement is not significant when the number of processors is increased. This can be explained by the fact that the central queue is a shared resource between the processes executing during one simulation cycle, and thus becomes a bottleneck when the number of processors increases. This leads us to conclude that central queue algorithms would be better suited for shared-memory multiprocessor systems of 8-10 processors. A prototype parallel simulator using the central event queue algorithm, developed by Vantage Systems [36] has demonstrated an average execution time speedup of 4 over a sequential simulator, on an 8-node symmetric multiprocessor system.



**Figure 5.1** Performance of a centralized-queue algorithm on fat-trees of different sizes.

The performance of distributed time algorithms is now considered. Figure 5.2 shows the speedup expected when a distributed-time algorithm is used on a fat-tree network, with varying numbers of processing nodes. The communication cost and rollback cost have been neglected at this stage. It is observed that the results obtained with distributed-time algorithms are superior to the results obtained with centralized-time algorithms. However, the roll-back cost and the communication costs have to be included in the tests before the distributed-time approach can be considered to be feasible.

Algorithm	Program	Speedup with 4 processors	Speedup with 8 Processors	Speedup with 16 Processors	Speedup with 32 processors
	Testbench 1	2.25	3.0	4.25	5.6
Central-queue	Testbench 2	1.3	1.4	1.44	1.5
	Testbench 3	1.5	1.66	1.75	1.56
	Testbench 1	2.56	3.57	7.33	11.58
Distributed-time	Testbench 2	2.25	3.85	5.35	6.4
	Testbench 3	2.62	5.25	5.95	6.0

**Table 5.2** Results obtained with Centralized and Distributed-time algorithms (rollback cost and communication cost are not considered)

The degradation in performance due to roll-back is taken into account at this stage. In order to perform a rollback, a process must save it's state from time to time, and must also keep a list of all messages it has received, so that it can process them after rolling back. Furthermore, the process must also keep a record of event messages it has sent to other processes, since some of them may result from execution which is subsequently rolled back. Figure 5.3 shows the performance of an optimistic distributed time algorithm when the cost of performing a rollback is taken into account. The following assumption is made about the rollback cost with respect to the execution time of a VHDL statement. A rollback cost of 100, for example, corresponds to the execution time of 100 VHDL statements. It is observed from the results obtained that the performance declines substantially when the rollback cost exceeds 100.

The communication latency of the interconnection topology is now examined in detail. The communication cost of messages being passed between processing nodes is of vital importance when the

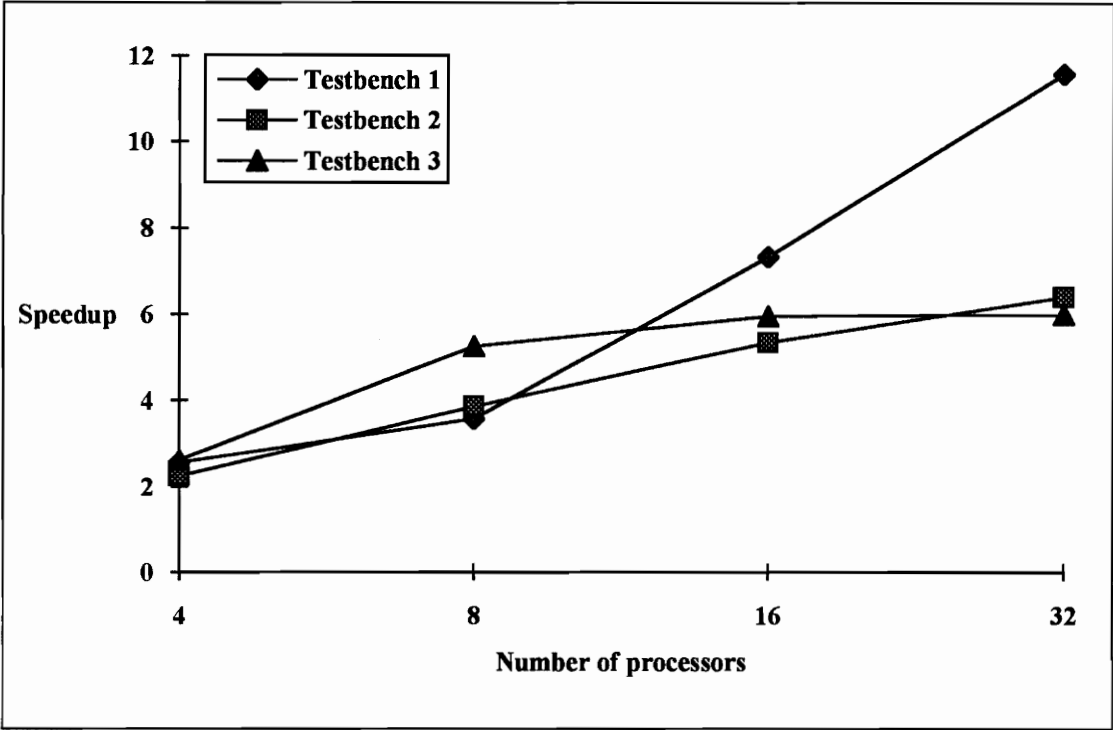
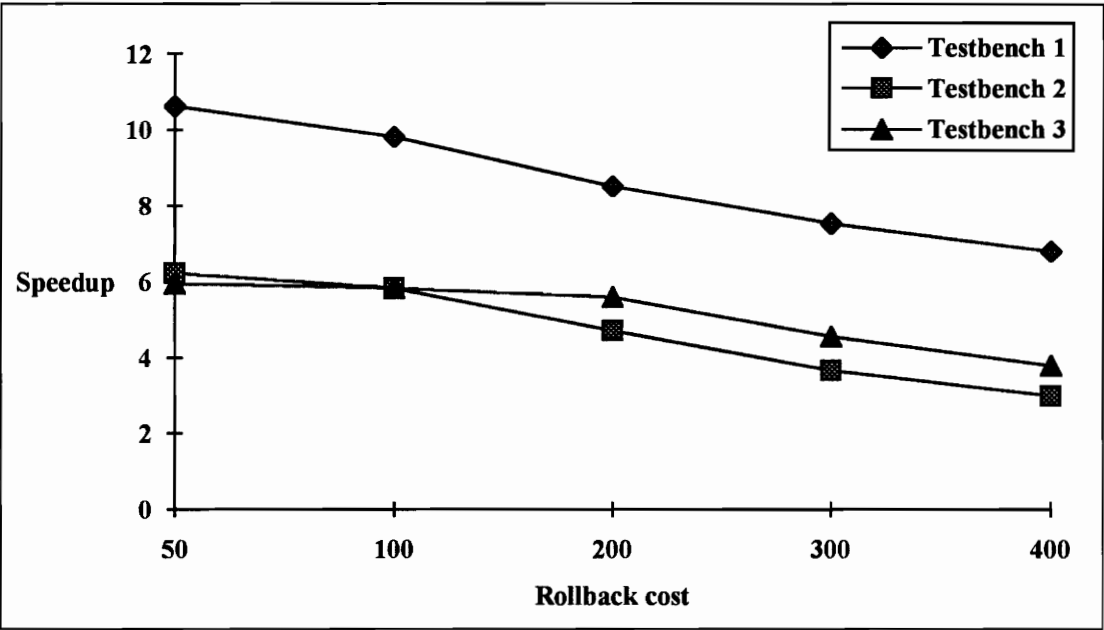


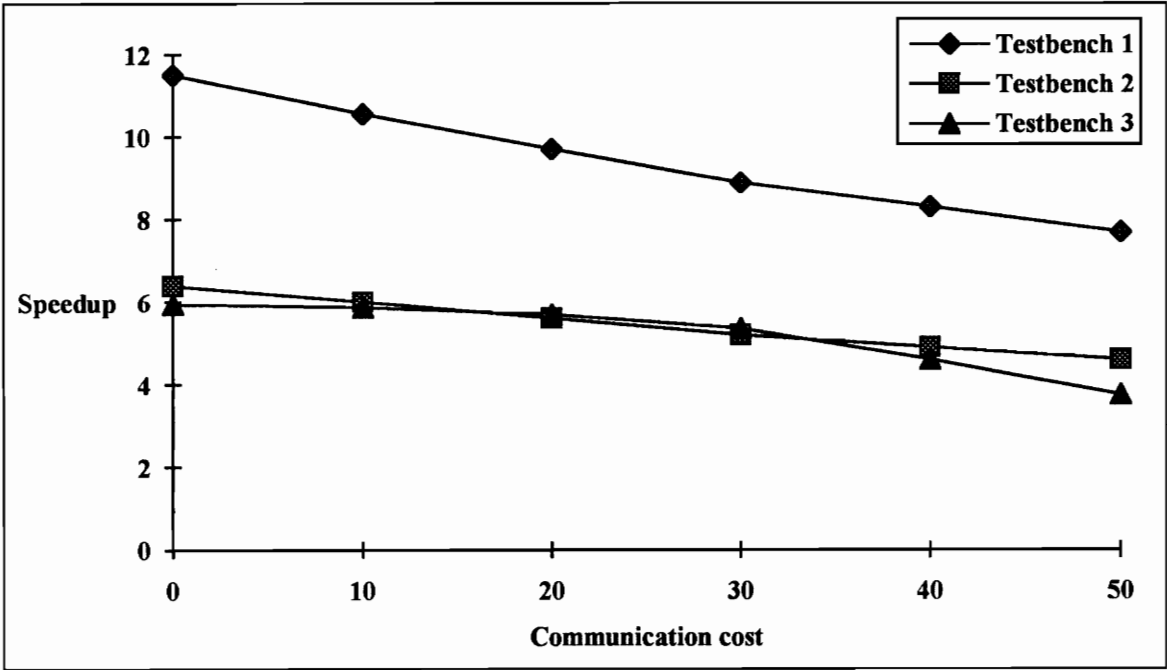
Figure 5.2 Performance of a distributed-time algorithm on fat-trees of different sizes.



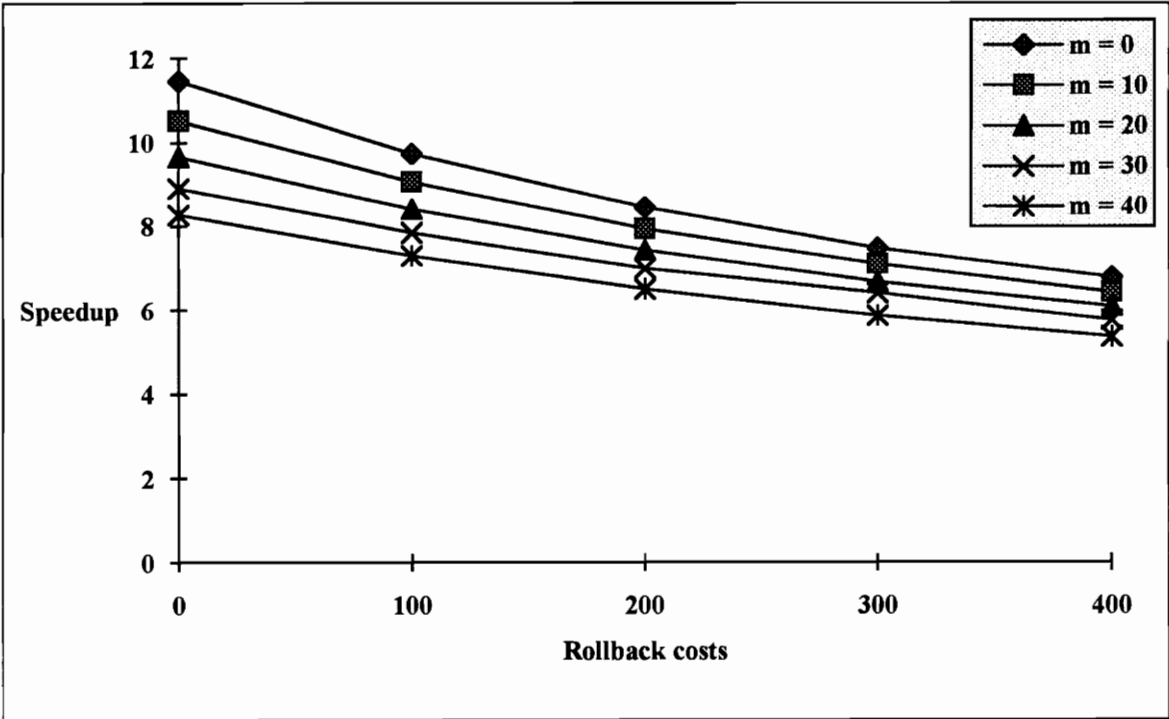
**Figure 5.3** Performance of an optimistic distributed-time algorithm of a 32-node fat-tree with increasing rollback costs.

overall performance of a specific network topology is to be determined. The maximum number of nodes that must be traversed in order to transmit a message between any source/destination processor pair should be as small as possible for the communication latency to be small. The communication cost is examined for 4, 8, 16 and 32-node fat-tree multicomputer networks. The following assumption is used to estimate the communication cost in relation to the execution time of a VHDL statement. A communication cost of 10 implies that the time taken to send an event message is comparable to the time taken to execute 10 VHDL statements. Figure 5.4 shows the results obtained by varying the communication costs for systems of various sizes. We can see that the communication latency causes the performance to decline beyond a threshold limit of about 10 after which most of the processor's execution time is spent waiting for messages. Figure 5.5 show the performance of an optimistic distributed-time algorithm for a 32-node fat-tree with increasing communication and rollback costs, using Testbench 1 as input. The communication cost ( $m$ ) is increased in multiples of 10 while the rollback cost is increased in multiples of 100. It is observed that the rate of performance degradation with increased rollback costs (for a given value of  $m$ ) is less than the rate of performance decline when the rollback cost is low.

A comparative analysis of the fat-tree topology versus the linear array topology is performed, by evaluating the performance of an optimistic distributed algorithm on a linear array of processors. The linear array topology was specifically chosen to provide a contrast to the fat-tree topology, as it is much simpler and less expensive to implement. However, the diameter of a linear array increases linearly with respect to the number of nodes, and hence the topology is not considered to "scale" well for a large number of processors [6]. Figure 5.6 shows the speedup obtained for linear arrays of varying sizes with varying communication costs factored into the evaluation. It can be seen that the decline in speedup is much more noticeable in the case of the linear array as compared to the fat-tree topology, especially when the communication cost exceeds 10. Figure 5.7 shows a comparison between a 32-node fat-tree and a 32-node linear array with increasing communication costs. The average values obtained from evaluating the three testbench programs (Figures 5.4 and 5.6) have been plotted in the graph to compare the two

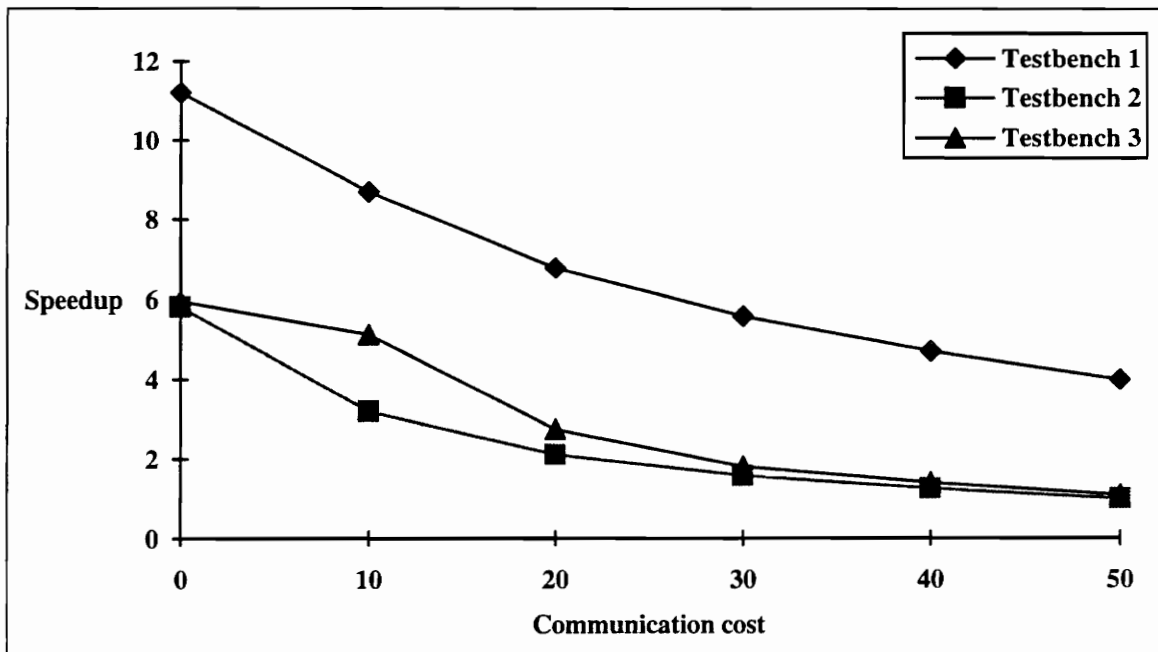


**Figure 5.4** Performance of an optimistic distributed-time algorithm of a 32 processor fat-tree with increasing communication costs.

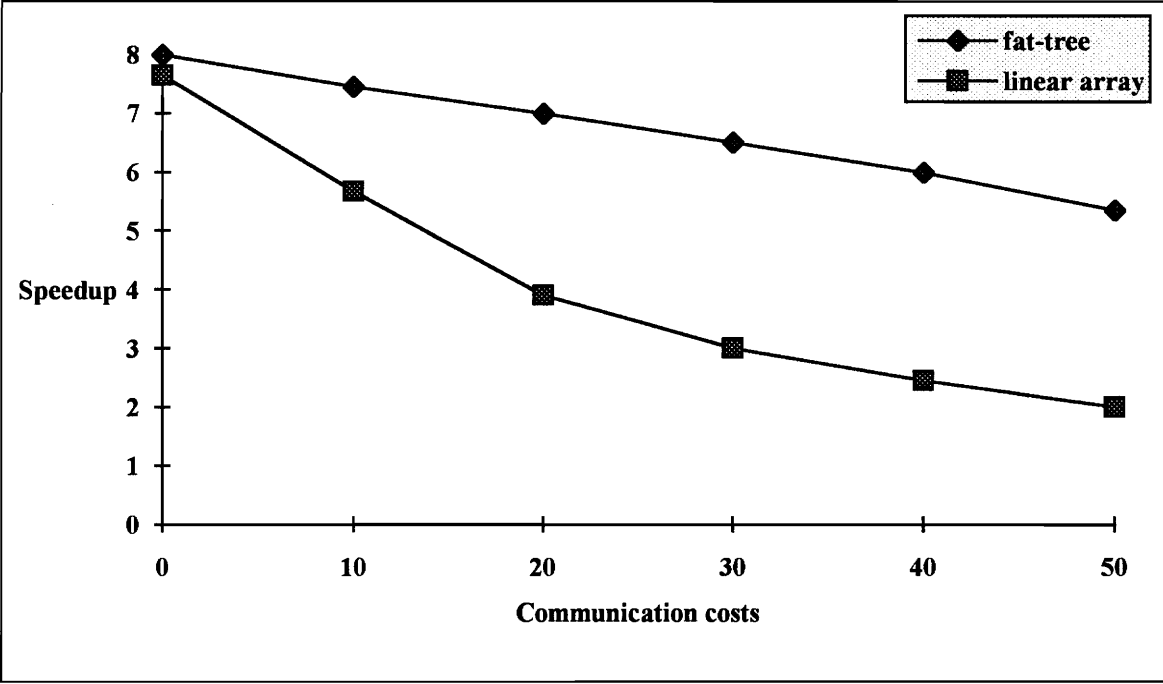


**Figure 5.5** Performance of an optimistic distributed-time algorithm of a 32-node fat-tree with increasing communication and rollback costs.





**Figure 5.6** Performance of an optimistic distributed-time algorithm of a 32 processor linear array with increasing communication costs.



**Figure 5.7** Comparison of the performance decline between a 32-node fat-tree and a 32-node linear array with increasing communication costs.

topologies. It took an average of 9 minutes to obtain each individual result, totaling 15 hours of simulation time (on a Sun Sparkstation 10) to obtain the results plotted in the graphs.

In order to gain a better insight into the differences in performance of the two topologies under consideration, the following mathematical analysis is carried out. The capacity for message passing in a binary fat tree with  $n$  processing nodes and  $n-1$  switching nodes may be measured and compared to the capacity of a linear array, by considering the case in which each switching node transfers messages to all other nodes in the tree.

Assume a binary fat-tree with  $n = 2^{k+1} - 1$  nodes. A node  $i$  at level  $p$  ( $1 \leq p \leq k+1$ ) is the root of a tree containing  $2^{k-p+2} - 1$  nodes, with  $2^{k-p+1} - 1$  nodes in its left and right subtrees, respectively.

There are  $(2^{k+1} - 1) - (2^{k-p+2} - 1) = 2^{k-p+2} (2^{p-1} - 1)$  nodes outside this subtree rooted in  $i$ .

The messages that are transferred by  $i$  are those that are:

- i) sent from a node in the subtree rooted in  $i$  (not including  $i$  itself) to a node outside this subtree or vice versa; there are

$$2 \cdot (2^{k-p+2} - 2) 2^{k-p+2} (2^{p-1} - 1) = (2^{k-p+1} - 1) 2^{k-p+4} (2^{p-1} - 1) \text{ of these messages;}$$

- ii) or sent from a node in the left subtree of  $i$  to a node in the right subtree of  $i$  or vice versa; there are

$$2 \cdot (2^{k-p+1} - 1) (2^{k-p+1} - 1) = 2 (2^{k-p+1} - 1)^2$$

Therefore, the total number of messages transferred by a node  $i$  at level  $p$  is

$$\text{trns}(p) = 2^{k+1} (2^{k-p+3} - 3(2^{k-2p+2} + 1) + 2^{p+3}) - (2^{k+1} - 2)$$

and hence the total number of messages transferred by the binary fat-tree is

$$\begin{aligned} \sum 2^{p-1} \text{trns}(p) &= 2^{2(k+1)} (2(k+1) - 7) + 2^k (2(k+1) + 9) - 2 - (2^{k+1} - 2) \\ &= (n+1)^2 (2(k+1) - 7) + (n+1)(2(k+1) + 9) - 2 - (n-1) \\ &= O(n^2 \log n) \end{aligned}$$

and the average number of messages transferred per node is  $O(n \log n)$ .

The same analysis applied to the linear array with  $n$  nodes shows that the  $i$ th node ( $1 \leq i \leq n$ ) receives  $n - 1$  messages directed to node  $i$  itself and  $2(i - 1)(n - i)$  messages from a node to the left of  $i$  to a node to the right of  $i$ , or vice versa.

Hence, the total number of messages transferred in a linear array is

$$\begin{aligned} \sum 2(i - 1)(n - i) &= n(n - 1)(n - 2)/3 \\ &= O(n^3) \text{ and, on the average, a node transfers } O(n^2) \text{ messages.} \end{aligned}$$

A comparison of the average number of message transfers per node of a binary fat-tree [ $O(n \log n)$ ] and the linear array [ $O(n^2)$ ] shows that the nodes in the fat-tree transfer less messages. This fact helps to explain the decline in performance of a linear array topology when the communication cost is considered.

The speedup of simulating the VHDL benchmark models, observed with the optimistic distributed parallel algorithms on a 32-node fat-tree is an order of magnitude greater than would be obtained with a sequential simulator running on a uniprocessor machine. Optimistic synchronization techniques, using the Time Warp approach, can be enhanced by dynamically adjusting the simulation parameters at run time to improve the probability that productive work is accomplished. For example, "bounded time windows," which are dynamically sized, can be used to improve the "look-ahead" mechanism in the Time Warp approach. This approach is currently under investigation at the University of Cincinnati [35].

## 5.5 VHDL Language Extensions for System-level Modeling

As the VHDL language continues to evolve with the addition of new features for system-level modeling, it is necessary to analyze their impact on the fat-tree interconnection network, proposed for the parallel simulation of the VHDL models. An issue which is not completely resolved so far is the inclusion of "shared variables" into the language (VHDL '93). The requirement for shared variables is due to the need to share data between processes, without the need for explicit modeling of the access to the shared data. There are various mechanisms used to provide "mutual exclusion" for the shared resources. Mutual exclusion ensures that a sequence of statements is treated as an indivisible operation. A sequence of

statements that must appear to be executed as an indivisible operation is called a "critical section." The term mutual exclusion refers to mutually exclusive execution of critical sections of code.

Three well-known approaches used to provide mutual exclusion for shared resources are *semaphores*, *critical regions* and *monitors*. With semaphores, the entry and exit from the critical sections are encapsulated by a pair of high-level primitives that disallow the critical section to be interrupted by other processes during their operation [7]. Critical regions can be implemented with the use of VHDL *access* statements, which define the critical section of code for one or more shared variables. The shared variables can be declared in VHDL architectures or blocks that enclose processes, and the processes may refer to the shared variables using access statements as shown below:

```
architecture proc of e is
    shared variable SV1 ; integer := 0;
begin
    p1 : process
    begin
        access (SV1)
            SV1 := SV1 +1;
        end access;
        wait on Sig1;
    end process P1;
end proc;
```

The use of critical regions for mutual exclusion is seen to restrict the nesting of access statements, in order to prevent the occurrence of "deadlock" amongst the processes. Furthermore, "information hiding," a key feature in VHDL, would be very difficult to implement with critical regions. These drawbacks prevent the adoption of the critical region mechanism, from inclusion into the language.

Monitors group shared data with the operations to be performed on that data [7]. With monitors, an implicit "lock" and "release" mutual exclusion mechanism is provided, based on entry to, and exit from, the monitor operations. An example has been provided below of a database application, to demonstrate the use of the monitor type in VHDL [37]. First a monitor is defined. Then, instances of the shared

variables using the monitor are created. Finally, operators defined by the monitor are applied to the shared variable.

```
type TreeT is monitor
  function add(t: TreeT; e: ElementT) return boolean;
  procedure merge(t1, t2: in TreeT; r: out TreeT);
end monitor TreeT;
```

```
type TreeT is monitor body
  function add(t: TreeT; e: ElementT) return boolean
    begin .... end;
  procedure merge(t1, t2: in TreeT; r: out TreeT)
    begin ..... end;
end monitor TreeT;
```

In the declarative part of an architecture the following declarations could be made:

```
shared variable a, b, c: TreeT;
variable e; ElementT;
```

In the sequential statement part of a process body the following statements could be found:

```
success = add(a,e);      -- monitor instance a implicitly locked during addition
merge(a,b,c);           -- monitor instances a, b and c implicitly locked during merge
```

A simulator using a distributed PDES algorithm might have different VHDL processes requesting access to shared variables, with the processes being at different simulation times. In order to correctly implement the language semantics, the simulator must synchronize processes that use shared variables, so that accesses occur in the correct order in simulation time. The monitor would encapsulate each shared variable and be responsible for granting access to one process at a time.

The future inclusion of shared variables into the language could have an important consequence on the parallel compilers running on multicomputers which use message-passing to communicate data between messages. The major choices for the communication method are message-passing and *distributed shared memory* (DSM). Most of the large scale multicomputers built to-date have used the message-passing model for communication due to the fact that they are simpler to build than the shared memory model. However, with the message-passing model, the compiler (or programmer) has to partition a program into separate processes that communicate explicitly, by sending messages rather than implicitly

through memory. In contrast, the DSM model allows the compiler to directly reference data in any of the physically distributed memories, independent of the location of the data. Furthermore, hardware-supported mechanisms used to communicate data amongst processors using the DSM model have a much smaller overhead than message-passing communication [9]. Taking into consideration the future inclusion of shared variables into the language, the DSM model would be a suitable communication method for the fat-tree multicomputer network as it would significantly reduce the complexity of developing parallel simulators for these machines.

## 5.6 Conclusions

This chapter presents research conducted on the parallel discrete event simulation of VHDL models. A multicomputer, based on the fat-tree interconnection network, is proposed as a suitable architecture for the parallel simulation. Parallel simulators, using optimistic distributed time algorithms offer the best opportunity to exploit the inherent parallelism available in VHDL models.

The Parallelism Analyzer was used to help evaluate the feasibility of this approach. Simulation outputs obtained from Chapter 4, where a VHDL model of the fat-tree was simulated, were used by the Parallelism Analyzer. The performance improvement obtained by partitioning the VHDL code of various benchmark programs amongst multiple processors of the fat-tree was found to be an order of magnitude greater than if the same program were to be simulated on a uniprocessor.

A comparative analysis was carried out between the fat-tree interconnection network topology and the linear array topology. It was observed that there is a greater performance decline in the case of the linear array, when the communication costs of passing messages amongst the processors is increased. This intuitive result was confirmed with the tests performed, as well as with a mathematical analysis carried out on the average number of messages transferred per node in the network.

A new feature, the "shared variable," which will be included in a future implementation of the VHDL language is analyzed along with the mechanisms used to provide mutual exclusion. The

distributed shared memory model, used for communication between the processors in the multicomputer would make the implementation of parallel simulators easier, and is proposed as a possible future enhancement in a multicomputer system used for the parallel simulation of VHDL models.



## Chapter 6

### Conclusions

There are a number of research programs in the area of high-performance computer systems presently being pursued at various universities and research organizations to analyze and evaluate the future directions of various computer architectures. A few of the issues currently being investigated at various organizations are:

- Should efforts be concentrated primarily on building more powerful Cray-type uniprocessor systems to help solve our future computational needs? A prototype of the Cray-4 vector supercomputer using GaAs technology is expected in late 1994 [1]. This approach appears to be feasible for the short term, but long-term demands of greater computing power would not be realizable with a uniprocessor, due to the speed-of-light barrier.
- Would linking as many uniprocessors together as are feasible, via a bus, crossbar or reconfiguring network provide us with scaleable computer systems for our future needs? A number of commercially available systems today use this approach, including the Sequent Symmetry S-81 system which connects up to 30 processors on a single bus [2]. Cost and performance issues of the links connecting the processors limit this option to a small number of processors, and hence this approach cannot be successfully scaled to solve problems using massively parallel algorithms.
- How suitable is the approach to use an interconnection network to link together a large number of processors and thereby provide us with general-purpose scalable computer systems? With such systems, what are the most appropriate topologies and allocations of resources to processor, memory, switches, and controllers? In this thesis, the fat-tree interconnection network topology was evaluated as a

multicomputer network for the parallel simulation of VHDL models. Communication issues related to the interconnection network were analyzed and algorithms used for discrete event simulation compared so as to determine which algorithms are best suited for use with parallel simulators over multicomputer networks.

## **6.1 Summary of Research Work**

In Chapter 2, an overview of multicomputer networks was presented. A classification of parallel processing architectures was provided along with performance criteria and design issues related to the various interconnection network topologies. Communication and synchronization issues of MIMD systems were analyzed. The factors affecting the scalability of parallel processing systems were then discussed. The chapter concluded with an overview of factors used to determine how performance characteristics scale with the number of processors being interconnected, using a specific network topology. The fat-tree interconnection topology was considered to be suitable for further investigation since it could be used to connect very large numbers of processing nodes with good performance characteristics.

Chapter 3 described the VHDL model of the processing node used in the fat-tree multicomputer network. The structural and behavioral level VHDL models used are based on the DLX processor which is a 32-bit reduced instruction set processor [3]. Since parallel algorithms make efficient use of memory hierarchies, the design included a cache memory in each processing node. A test program explored instruction level parallelism. Unfortunately, instruction-level parallelism can only be exploited to a limited extent due to the constraints of program behavior, compiler and operating system limitations and evaluation mechanisms built into modern computer systems. A solution to this problem is to link together a large number of processing nodes using a multicomputer network. The problem to be solved is distributed among the available multiple processing nodes and thereby a speedup in the program execution time occurs.

Chapter 4 presented research dealing with aspects of effectively modeling multicomputers using the VHDL language. Multicomputer networks based on simple tree structures have been constructed at various research organizations. When many messages have to pass through the root-node, the channel through which the information would have to pass quickly becomes congested. Tree-based structures had limited success when they were used to solve general-purpose problems, and thus they were used mainly to solve a few special-purpose algorithms.

The binary fat-tree architecture is able to alleviate the root congestion problem by increasing the channel capacity nearer to the root. The *Universality theorem* is used to show that an N-node fat-tree (but not 2-D arrays or ordinary trees) can simulate any other N-node structure of the same volume in, at worst  $O(\log^3 N)$  time. The VHDL model of the fat-tree with behavioral-level processing nodes is used to run sample programs and the dynamic execution count of instructions measured. Interestingly, this mix is observed to be different in the case of parallel programs as compared to sequential programs, with fine-grained parallel programs being more control-intensive than sequential programs. This result indicates that it would be beneficial to build special hardware mechanisms into the processing nodes to improve the message-transfer latency between the nodes.

Large VHDL models take a very long time to simulate as since they are computationally intensive and they utilize a large portion of the systems available memory. The computer system resources used for the simulation (Sun Sparkstation 2 running SunOS 4.1.3 with 32 MB RAM) were carefully monitored as the size of the fat-tree model was increased from 4 processing nodes to 32 processing nodes and a threshold level was determined after which simulation efficiency significantly declined. Successively larger models were created and the system resources monitored during simulation runs to determine where bottlenecks usually occur. It was observed that the run time increased greater-than-linearly with an increase in the number of processing nodes. Figure 4.5 shows that it took 2.3 minutes to simulate a 4-node fat-tree network. To simulate a 16-node network the run time increased by a factor of 4.78 to 11.1 minutes. When the number of processing nodes was increased to 32, the run time nearly tripled to 30.5

run time was observed, especially when the amount of memory paging activity significantly increased, as in the case of the 32-node fat-tree. Chapter 4 thus dealt with the efficient simulation of large VHDL models on a uniprocessor workstation. To reduce the time taken for the simulation of large VHDL models, it was proposed to exploit the parallelism inherent in VHDL by distributing the simulation of complex models over multiple processors in a multicomputer network.

Chapter 5 presents research on the parallel simulation of VHDL models over a fat-tree multicomputer network. A Parallelism Analyzer (PA) was used to help evaluate the feasibility of this approach. Simulation outputs obtained from simulating the fat-tree model (in Chapter 4) were used by the analyzer to carry out the tests. The behavior of a distributed VHDL simulation was emulated on a uniprocessor workstation, by analyzing the events of a sequential simulation and mapping the sequential events to the corresponding processors, while ensuring that the dependency constraints were satisfied.

On the basis of the simulation results obtained, it appears that parallel simulators using optimistic distributed time algorithms offer the best opportunity to exploit the inherent parallelism available in VHDL models. The performance improvement obtained by partitioning the VHDL source code of benchmark programs amongst multiple processors of the fat-tree was found to be an order of magnitude greater than if the same programs were to be simulated on a uniprocessor. The communication cost of passing messages was examined for 4, 8, 16 and 32-node fat-tree multicomputer networks. It was observed that the communication latency causes the performance to decline beyond a threshold limit of about 10, after which most of the processor's execution time is spent waiting for messages. Furthermore, the cost of performing a rollback was also considered for an optimistic distributed time algorithm. It was seen that the performance declines substantially when the rollback cost exceeds 100.

To gain a clearer insight into the results obtained, a comparative analysis was carried out between the fat-tree interconnection network topology and the linear array topology. There was a greater performance decline in the case of the linear array, when the communication cost of passing messages amongst the processors was increased. This result was confirmed with the tests performed, as well as with

a mathematical analysis carried out on the average number of messages transferred per node in the network. A comparison of the average number of message transfers per node of a binary fat-tree [ $O(n \log n)$ ] and the linear array [ $O(n^2)$ ] shows that the nodes in the fat-tree transfer less messages.

A new feature, the "shared variable," to be included in a future implementation of the VHDL language was analyzed along with the mechanisms used to provide mutual exclusion. The distributed shared memory model, used for communication between the processing nodes in the fat-tree network would make the implementation of parallel simulators easier, and was proposed as a future enhancement in multicomputer systems used for the distributed simulation of VHDL models.

## 6.2 Future Work

The research conducted in this thesis can be expanded in the following areas:

- i) The results obtained in Chapter 5 indicate that an order of magnitude increase in performance of the execution time of a VHDL model simulation can be expected by simulating the model over a multicomputer network. A parallel simulator needs to be designed and implemented using distributed-time algorithms on real systems (for example, the Connection Machine-5) which uses the fat-tree architecture.
- ii) Optimistic distributed algorithms based on the Time Warp paradigm can be further optimized. When a process rolls back, instead of immediately canceling events it has generated, it may first check to see if the new events it generates are the same as the old ones. If this is the case, the original events need not be canceled.
- iii) The assignment of discrete simulation tasks to different processors in a message-passing multicomputer is a complex issue involving load balancing, inter-processor communication, and intra-processor communication. Finding an optimal processor mapping is an NP-complete problem and heuristics are used to get a near-optimal solution. Using knowledge about the internal hierarchy of a

simulated system, the simulation tasks may be partitioned into clusters such that the tasks in the same cluster are assigned to the same processor.

## Appendix A

A sequential program to compute prime numbers using the Sieve of Erathosthenes is shown below. In order to compute the prime numbers between 1 and  $n$ , an  $n$ -element boolean array is used. Initially, the program initializes all the array elements to the value '1' (or true). The remainder of the program changes these array elements from 1 to 0 whenever a number is found to be non-prime. At the end of the program, all of the remaining elements of the array which still have the value '1' are the prime numbers between 1 and  $n$ .

```
    ; program to find prime numbers
    .text    0
    ADD     R1, R0, X2
    ADD     R2, R0, X3
    LW      R3, 0(R1)
    LW      R4, 0(R2)
Loop:  ADD     R6, R5, R4
       SW      0(R6), R7
       ADD     R7, R7, #8
       SUB     R4, R6, R5
       ADD     R5, R6, R0
       SLT     R8, R6, XTOP
       BNEZ    Loop
       TRAP    #0
```

The first prime number identified is the prime number 2. After this, all the multiples of this prime are marked as being non-prime. The program then finds the next prime number and marks them as being non-prime. This process continues until the multiples of all the primes have been marked as being non-prime numbers.

The above program can be parallelized by partitioning the array into equal-size portions and creating one parallel process to work on each process. If the grain size of each sub-task is reduced and distributed among a greater number of processors, the amount of computation performed by each processor decreases. A massively parallel version of the program using  $n-1$  processors can be implemented by having each processing element (PE) represent an integer between 1 and  $n$ . Each PE

remains active as long as its number is a potential prime number. In each iteration of the program, the smallest number which is still active is marked off as a prime and all the remaining PEs test to see if the number that they represent is a multiple of the prime number. In this manner, the remaining multiples of the prime numbers are eliminated in parallel.



## Bibliography

- [1] John Hennesy and David A. Patterson, *Computer Architecture: A Quantitive Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.
- [2] Kai Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.
- [3] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Computers*, Vol. 21, No. 9, pp. 948-960, 1972.
- [4] Harold W. Lawson, *Parallel processing in industrial real-time applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.
- [5] Angel L. DeCegama, *Parallel processing architectures and VLSI hardware*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.
- [6] Thomas Braunl, *Parallel Programming: an introduction*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1993.
- [7] George Almasi and Allan Gottlieb, *Highly Parallel Computing*, 2nd Edition, The Benjamin Cummings Publishing Company Inc., Redwood City, California, 1994.
- [8] John T. McHenry, "Multicomputer Networks for Smart Structures," Ph.D. Dissertation, Department of Electrical Engineering, Virginia Tech, 1993.
- [9] T. Lang, M. Valero, and I. Alegre, "Bandwidth Analysis of Crossbar and Multiple-Bus Contentions for Multiprocess Programs," *IEEE Trans. Computers*, Vol. 31, No. 1, pp. 1227-1233, Jan. 1982.
- [10] Cray Research Inc., "Cray T3D: Technical Summary," September 1993.
- [11] L. Lamport, "Time, Clock, and Ordering Events in a Distributed System," *Communicationf of the ACM*, Vol. 21, No. 7, pp. 558-564, July 1978.

- [12] E. W. Dijkstra, "Cooperating Sequential Processes," in Genuys (ed.), *Programming Languages*, Academic Press, New York, 1968.
- [13] nCUBE, *nCUBE 6400 Processor Manual*, nCUBE Company, Beaverton, OR, 97006, 1990.
- [14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N. J. pp. 483-485, April, 1967.
- [15] Charles E. Leiserson, "Fat-trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892-901, October 1985.
- [16] DLX Processor model, Obtained from Anonymous FTP site at The University of Adelaide (chook.cs.adelaide.edu.au), Australia, February, 1994.
- [17] Forest Baskett and John Hennesy , "Microprocessors: From Desktops to Supercomputers," *Science*, Vol. 261, pp. 864-871, August 1993.
- [18] John P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, Inc., New York, 1978.
- [19] G. Goos and J. Hartmanis (Ed.), *Lecture Notes in Computer Science*, Springer Verlag, Berlin, Germany, 1983
- [20] Jean-Pierre Barthelemy and Alain Guenoche, *Trees and Proximity Representations*, John Wiley and Sons, New York, 1991.
- [21] Hosam M. Mahmoud, *Evolution of Random Search Trees*, John Wiley and Sons, New York, 1992.
- [22] Gyula Mago, "A Cellular Computer Architecture for Functional Programming," *COMPCON Proceedings*, pp. 178-187, February 25-28, 1980.
- [23] David E. Shaw, "SIMD and MSIMD Variants of the NON-VON Supercomputer," *COMPCON Proceedings*, pp. 360-363, March, 1984.
- [24] Salvatore J. Stolfo and Daniel P. Miranker, "The DADO Production System Machine," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 269-296, 1986.

- [25] C. E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffery V. Hill, Daniel Hillis, et al, "The Network Architecture of the Connection Machine CM-5," Technical paper, Thinking Machines Corporation, Cambridge, Massachusetts, April 1992.
- [26] Peter J. Ashenden, "Recursive and Repetitive Hardware Models in VHDL," Technical Report 93-19, Department of Computer Science, The University of Adelaide, Australia, 1993.
- [27] Ellen Spertus, Seth Goldstein, Klaus Schauser, Thorsten vo Eicken, David Culler, William Dally, "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," Technical Report, M.I.T., 1993.
- [28] Jean-Michel Berge, *VHDL Designer's Reference*, Kluwer Academic Publishers, 1993.
- [29] Mike Loukides, *System Performance Tuning*, O'Reilly and Associates, Inc., Sebastopol, California, 1990.
- [30] Peter J. Ashenden, Henry Detmold, Wayne S. McKeen, "Parallel Execution of VHDL Models," Technical Report 93-01, Department of Computer Science, The University of Adelaide, Australia, January 1993.
- [31] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33., No. 10, pp. 30-53, October, 1990.
- [32] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Communications," *Communications of the ACM*, Vol. 24, No. 4, pp. 198-206, April 1981.
- [33] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp 404-425, July 1985.
- [34] Alessandra Costa, et al, "An Evaluation System for Distributed-Time VHDL Simulation," Technical Paper, University of Genoa, Italy, 1993.
- [35] Venkat Krishnaswamy, "Quest Project," University of Cincinnati, May 1994.
- [36] Minutes of the Parallel VHDL Simulation Study Group, Computer Science Corporation, San Diego, California, June 1993.

- [37] John Willis, Rob Newshutz, Rita Glover, Steven Bailey, Victor Berman, Oz Levia, "Requirements Document for VHDL Shared Variables Revision: Version 4.6," Technical Report, May 1994.

## Vita

Vikrampal Chadha was born in Bombay, India in 1967. He received the Bachelor of Engineering degree in Electronics and Communications Engineering at the Manipal Institute of Technology, India in June 1990. In 1991, he began his graduate studies in the Bradley Department of Electrical Engineering at Virginia Tech. After two semesters as a student, he worked as a Software Engineer at Electronet Information Systems, Inc. as part of the Graduate Cooperative Education Program at Virginia Tech, during the 1992-93 year. The work reported in this thesis was conducted between January 1994 and September 1994. His research interests are in the areas of parallel processing, simulation and computer networks.

